Natasha Sharygina
Helmut Veith (Eds.)

LNCS 8044

Computer Aided Verification

25th International Conference, CAV 2013
Saint Petersburg, Russia, July 2013
Proceedings

Springer

Lecture Notes in Computer Science 8044

Commenced Publication in 1973
Founding and Former Series Editors:
Gerhard Goos, Juris Hartmanis, and Jan van Leeuwen

Natasha Sharygina Helmut Veith (Eds.)

Computer Aided Verification

25th International Conference, CAV 2013
Saint Petersburg, Russia, July 13-19, 2013
Proceedings

Springer

Volume Editors

Natasha Sharygina
University of Lugano, 6900 Lugano, Switzerland
E-mail: natasha.sharygina@usi.ch

Helmut Veith
Vienna University of Technology, 1040 Vienna, Austria
E-mail: veith@forsyte.at

ISSN 0302-9743 e-ISSN 1611-3349
ISBN 978-3-642-39798-1 e-ISBN 978-3-642-39799-8
DOI 10.1007/978-3-642-39799-8
Springer Heidelberg Dordrecht London New York

Library of Congress Control Number: 2013943285

CR Subject Classification (1998): D.2.4, I.2.2, D.2, F.3, F.1, F.4, C.3, D.1

LNCS Sublibrary: SL 1 – Theoretical Computer Science and General Issues

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Preface

Умом Россию не понять,
Аршином общим не измерить:
У ней особенная стать —
В Россию можно только верить.

Ф. И. Тютчев, 1866

You will not grasp her with your mind
Or cover with a common label,
For Russia is one of a kind —
Believe in her, if you are able...

F.I. Tyutchev, 1866
Translation, Anatoly Liberman

This volume contains the proceedings of the 25th International Conference on Computer Aided Verification (CAV) held in Saint Petersburg, Russia, July 13–19, 2013. The Conference on Computer Aided Verification (CAV) is dedicated to the advancement of the theory and practice of computer-aided formal methods for the analysis and synthesis of hardware, software, and other computational systems. Its scope ranges from theoretical results to concrete applications, with an emphasis on practical verification tools and the underlying algorithms and techniques.

In its 25th anniversary, CAV received an all-time record of 209 submissions, and accepted a record number of 54 regular and 16 tool papers for presentation. We appreciate the excellent work of our Program Committee and our external reviewers, and thank them for their efforts; we received 849 reviews, and there was intense discussion on the papers after the author response period. Following a recent tradition, CAV had four special tracks highlighted in the program: Hardware Verification, Computer Security, Biology, and SAT and SMT. We thank our Track Chairs Armin Biere, Somesh Jha, Nikolaj Bjoerner, Jasmin Fisher, and the Tool Chair Roderick Bloem for their effort in attracting papers and supporting the review process, and our Proceedings Chair Georg Weissenbacher for preparing this volume.

The 25th anniversary of CAV an opportunity to learn from experience and plan for the next decades. To this end, CAV 2013 featured a panel discussion on the future of our research area and of the conference with keynote talks by Ed Clarke and Bob Kurshan.

The conference included two workshop days, a tutorial day, and four days for the main program. In addition to the contributed talks, the conference featured three invited talks and four invited tutorials:

Invited Speakers

Jennifer Welch (Texas A&M University)
Challenges for Formal Methods in Distributed Computing

Jeannette Wing (Microsoft Research International)
Formal Methods from an Industrial Perspective

Maria Vozhegova (Sberbank)
Information Technology in Russia

Invited Tutorial Speakers

Cristian Cadar (Imperial College London)
Dynamic Symbolic Execution

David Harel (The Weizmann Institute of Science)
Can We Computerize an Elephant? On the Grand Challenge of Modeling a Complete Multi-Cellular Organism

Andreas Podelski (University of Freiburg)
Software Model Checking for People Who Love Automata

Andrei Voronkov (The University of Manchester)
First-Order Theorem Proving and Vampire

The main conference was preceded by eight affiliated workshops:

- 6th International Workshop on Exploiting Concurrency Efficiently and Correctly
- Second CAV Workshop on Synthesis
- Second International Workshop on Memory Consistency Models
- Interpolation: From Proofs to Applications
- Verification and Assurance
- Verification of Embedded Systems
- Verification and Program Transformation
- Fun with Formal Methods

We thank Igor Konnov, our Workshop Chair, for the efficient organization of the workshops. We also thank the members of the CAV Steering Committee — Michael Gordon, Orna Grumberg, Bob Kurshan, and Ken McMillan — for the excellent collaboration and their advice on various organizational matters.

The practical organization of the conference was made possible by the collaboration of many people in Saint Petersburg, Lugano, and Vienna. First of all, we thank our Organization Chair Irina Shoshmina as well as Igor Konnov for their tireless work and their enthusiasm. We further thank Yuri Karpov, Dmitry Koznov, Yuri Matiyasevich, Boris Sokolov, Tatiana Vinogradova, Nadezhda Zalesskaya, Katarina Jurik, Thomas Pani, Yulia Demyanova, Francesco Alberti, Antti Hyvärinen, Simone Fulvio Rollini, and Grigory Fedyukovich for their help on all levels of the organization. We are grateful to Andrei Voronkov for the use

of the EasyChair system, and to Alfred Hofmann and his team for the smooth collaboration with Springer.

We gratefully acknowledge the donations provided by our corporate sponsors — Microsoft Research, Facebook, Coverity, IBM, Jasper, NEC, Cadence, Intel, Monoidics, Springer — and are thankful to our Publicity Chair Hana Chockler for attracting a record amount of industrial donations. Finally, we thank our academic sponsors, the University of Lugano, the Vienna University of Technology, the St. Petersburg State Polytechnical University, the Russian Academy of Science, the St. Petersburg Department of the Steklov Institute for Mathematics, the Euler International Mathematical Institute, and The Kurt Gödel Society, whose staff supported us in the organization of CAV 2013.

May 2013

Natasha Sharygina
Helmut Veith

Organization

Program Committee

Rajeev Alur	University of Pennsylvania, USA
Domagoj Babic	University of California Berkeley, USA
Armin Biere	Johannes Kepler University Linz, Austria
Nikolaj Bjorner	Microsoft Research
Roderick Bloem	Graz University of Technology, Austria
Ahmed Bouajjani	LIAFA, University of Paris 7 (Paris Diderot), France
Aaron Bradley	University of Colorado Boulder, USA
Hana Chockler	King's College London, UK
Byron Cook	Microsoft Research, University College London, UK
Vincent Danos	University of Edinburgh, UK
Anupam Datta	Carnegie Mellon University, USA
Javier Esparza	Technische Universität München, Germany
Azadeh Farzan	University of Toronto, Canada
Bernd Finkbeiner	Saarland University, Germany
Jasmin Fisher	Microsoft Research
Radu Grosu	Vienna University of Technology, Austria
Arie Gurfinkel	Carnegie Mellon Software Engineering Institute, USA
William Hung	Synopsys Inc.
Somesh Jha	University of Wisconsin, USA
Susmit Jha	Intel Strategic CAD Labs
Barbara Jobstmann	EPFL, Jasper DA, CNRS-Verimag
Bengt Jonsson	Uppsala University, Sweden
Rajeev Joshi	Laboratory for Reliable Software, NASA JPL
Joost-Pieter Katoen	RWTH Aachen, Germany
Robert P. Kurshan	Cadence Design Systems
Benjamin Livshits	Microsoft Research
P. Madhusudan	University of Illinois at Urbana-Champaign, USA
Rupak Majumdar	Max Planck Institute for Software Systems
Ken McMillan	Cadence Berkeley Labs
Kedar Namjoshi	Bell Labs
Joël Ouaknine	Oxford University, UK
Corina Pasareanu	Carnegie Mellon University, NASA Ames, USA
Andreas Podelski	University of Freiburg, Germany

Natasha Sharygina	University of Lugano, Switzerland
Nishant Sinha	IBM Research Labs
Anna Slobodova	Centaur Technology
Murali Talupur	Intel
Stavros Tripakis	University of California Berkeley, USA
Helmut Veith	Vienna University of Technology, Austria
Mahesh Viswanathan	University of Illinois at Urbana-Champaign, USA
Thomas Wahl	Northeastern University
Georg Weissenbacher	Vienna University of Technology, Austria
Eran Yahav	Technion, Israel

Additional Reviewers

Albarghouthi, Aws
Aleksandrowicz, Gadi
Antonopoulos, Timos
Asarin, Eugene
Atig, Mohamed Faouzi
Auerbach, Gadiel
Basler, Gérard
Bau, Jason
Ben-Amram, Amir
Benton, Nick
Berdine, Josh
Beyene, Tewodros A.
Bogomolov, Sergiy
Bonnet, Remi
Bordeaux, Lucas
Borgström, Johannes
Botincan, Matko
Brockschmidt, Marc
Buchholz, Peter
Bundala, Daniel
Chadha, Rohit
Chaki, Sagar
Chatterjee, Krishnendu
Chaudhuri, Swarat
Chen, Hong Yi
Chen, Juan
Clarke, Edmund
D'Silva, Vijay
Dang, Thao
Davare, Abhijit
David, Yaniv
Davidson, Drew
De Carli, Lorenzo
De Moura, Leonardo
De Paula, Flavio
Dehnert, Christian
Delzanno, Giorgio
Deshmukh, Jyotirmoy
Dietsch, Daniel
Dimitrova, Rayna
Dong, Jin Song
Donzé, Alexandre
Doucet, Fred
Doyen, Laurent
Drachsler, Dana
Drăgoi, Cezara
Duggirala, Parasara Sridhar
Ehlers, Rüdiger
Eisner, Cindy
Emmi, Michael
Enea, Constantin
Esmaeilsabzali, Shahram
Etessami, Kousha
Eén, Niklas
Faymonville, Peter
Filieri, Antonio
Flener, Pierre
Fredrikson, Matt
Frehse, Goran
Fuhs, Carsten

Gaiser, Andreas
Galceran-Oms, Marc
Ganty, Pierre
Garg, Deepak
Garg, Pranav
Garoche, Pierre-Loic
Gerke, Michael
Gligoric, Milos
Goldberg, Eugene
Gretz, Friedrich
Griesmayer, Andreas
Gueta, Guy
Günther, Henning
Haase, Christoph
Habermehl, Peter
Hahn, Moritz
Haller, Leopold
Hamadi, Youssef
Han, Tingting
Harmer, Russ
Harris, William
Haziza, Frédéric
Heinen, Jonathan
Heizmann, Matthias
Herreira, Christian
Hetzl, Stefan
Heule, Marijn
Hilston, Jane
Ho, Hsi-Ming
Holzer, Andreas
Holík, Lukáš
Hönicke, Jochen
Ivrii, Alexander
Jackson, Ethan K.
Jacobs, Swen
Jain, Mitesh
Jansen, Christina
Jayaraman, Karthick
Jezequel, Loig
Jha, Sumit
Jin, Hoonsang
Kahlon, Vineet
Kassios, Ioannis
Katz, Omer
Katzenbeisser, Stefan
Kennedy, Andrew
Khalimov, Ayrat
Khlaaf, Heidy
Kidd, Nicholas
Kiefer, Stefan
Kim, Hyondeuk
Kincaid, Zachary
Kini, Dileep
Kodakara, Sreekumar
Komuravelli, Anvesh
Konnov, Igor
Kordy, Piotr
Kotek, Tomer
Kovács, Laura
Kovásznai, Gergely
Kowaliw, Taras
Kretinsky, Jan
Krivine, Jean
Krstić, Sava
Kuncak, Viktor
Kuo, Jim Huan-Pu
Kupferman, Orna
Kupriyanov, Andrey
Kuraj, Ivan
Kwiatkowska, Marta
Könighofer, Bettina
Könighofer, Robert
Laarman, Alfons
Le, Hoang M.
Lee, Choonghwan
Legay, Axel
Lenhardt, Rastislav
Leonardsson, Carl
Li, Guodong
Li, Xuandong
Li, Yi
Lindén, Jonatan
Liu, Peizun
Liu, Wanwei
Liu, Zhaoliang
Luchaup, Daniel
Luttenberger, Michael
Mai, Haohui
Mandrali, Eleni
Margalit, Oded

Marron, Mark
Meshman, Yuri
Meyer, Roland
Moarref, Salar
Morgenstern, Andreas
Müller, Peter
Narayan Kumar, K.
Neumann, René
Nghiem, Truong
Niksic, Filip
Nimkar, Kaustubh
Noll, Thomas
Norman, Gethin
Novikov, Sergey
Nutz, Alexander
Papavasileiou, Vasilis
Parker, David
Parkinson, Matthew
Parthasarathy, Ganapathy
Partush, Nimrod
Pearson, Justin
Peled, Doron
Perdrix, Simon
Pereszlényi, Attila
Phan, Linh
Piskac, Ruzica
Piterman, Nir
Popeea, Corneliu
Prakash, Ravi
Qadeer, Shaz
Qian, Kairong
Rabe, Markus N.
Raghothaman, Mukund
Ramachandran, Jaideep
Ramalingam, Ganesan
Ray, Sayak
Raz, Orna
Rezine, Othmane
Riener, Heinz
Rinetzky, Noam
Ristenpart, Tom
Roohi, Nima
Rosu, Grigore
Rozier, Kristin Yvonne
Rungta, Neha
Rybalchenko, Andrey
Rümmer, Philipp
Sangnier, Arnaud
Sankaranarayanan, Sriram
Sapra, Samir
Schallhart, Christian
Schlund, Maximilian
Schnoebelen, Philippe
Schwartz-Narbonne, Daniel
Schwoon, Stefan
Seidl, Martina
Sethi, Divjyot
Sezgin, Ali
Sighireanu, Mihaela
Siminiceanu, Radu
Somenzi, Fabio
Song, Xiaoyu
Spieler, David
Stergiou, Christos
Stigge, Martin
Stoelinga, Marielle I.A.
Strichman, Ofer
Tabaei, Mitra
Tabuada, Paulo
Tautschnig, Michael
Thakur, Aditya
Thompson, Sarah
Tiwari, Ashish
Tkachuk, Oksana
Torfah, Hazem
Trivedi, Ashutosh
Tschantz, Michael
Tögl, Ronald
Vardi, Moshe
Varghese, Thomas Methrayil
Vechev, Martin
Velev, Miroslav
Villard, Jules
Vizel, Yakir
Vojnar, Tomáš
von Essen, Christian
von Gleissenthall, Klaus
Wachter, Björn
Wang, Zilong
Westphal, Bernd

Widder, Josef
Wintersteiger, Christoph
Wolf, Verena
Worrell, James
Yao, Penghui
Yenigun, Husnu
Yi, Wang
Yorav, Karen
Yorsh, Greta
Yuan, Jun
Zanghellini, Jurgen
Zhu, Charlie Shucheng
Zuleger, Florian
Zuliani, Paolo

Table of Contents

Invited Tutorials

Biology

Concurrency

Hardware

Hybrid Systems

Interpolation

Loops and Termination

New Domains

Probability and Statistics

SAT and SMT

Security

Shape Analysis

Software Verification

Synthesis

Time

First-Order Theorem Proving and VAMPIRE*

Laura Kovács[1] and Andrei Voronkov[2]

[1] Chalmers University of Technology
[2] The University of Manchester

Abstract. In this paper we give a short introduction in first-order theorem proving and the use of the theorem prover VAMPIRE. We discuss the superposition calculus and explain the key concepts of saturation and redundancy elimination, present saturation algorithms and preprocessing, and demonstrate how these concepts are implemented in VAMPIRE. Further, we also cover more recent topics and features of VAMPIRE designed for advanced applications, including satisfiability checking, theory reasoning, interpolation, consequence elimination, and program analysis.

1 Introduction

VAMPIRE is an automatic theorem prover for first-order logic. It was used in a number of academic and industrial projects. This paper describes the current version (2.6, revision 1692) of VAMPIRE. The first version of VAMPIRE was implemented in 1993, it was then rewritten several times. The implementation of the current version started in 2009. It is written in C++ and comprises about 152,000 SLOC. It was mainly implemented by Andrei Voronkov and Krystof Hoder. Many of the more recent developments and ideas were contributed by Laura Kovács. Finally, recent work on SAT solving and bound propagation is due to Ioan Dragan.

We start with an overview of some distinctive features of VAMPIRE.

- VAMPIRE is *very fast*. For example, it has been the winner of the world cup in first-order theorem proving CASC [32,34] twenty seven times, see Table 1, including two titles in the last competition held in 2012.
- VAMPIRE runs on all common platforms (Linux, Windows and MacOS) and can be downloaded from `http://vprover.org/`.
- VAMPIRE can be used in a *very simple way* by inexperienced users.
- VAMPIRE implements a unique *limited resource strategy* that allows one to find proofs quickly when the time is limited. It is especially efficient for short time limits which makes it indispensable for use as an assistant to interactive provers or verification systems.
- VAMPIRE implements *symbol elimination*, which allows one to automatically discover first-order program properties, including quantified ones. VAMPIRE is thus the first theorem prover that can be used not only for proving, but also for generating program properties automatically.

* This research is partially supported by the FWF projects S11410-N23 and T425-N23, and the WWTF PROSEED grant ICT C-050. This work was partially done while the first author was affiliated with the TU Vienna.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 1–35, 2013.

Table 1. VAMPIRE's trophies

CASC	year	FOF	CNF	LTB
CASC-16, Trento	1999		✔	
CASC-17, Pittsburgh	2000	✔		
CASC-JC, Sienna	2001		✔	
CASC-18, Copenhagen	2002	✔	✔	
CASC-19, Miami	2003	✔	✔	
CASC-J2, Cork	2004	✔	✔	
CASC-20, Tallinn	2005	✔	✔	
CASC-J3, Seattle	2006	✔	✔	
CASC-21, Bremen	2007	✔	✔	
CASC-J4, Sydney	2008	✔	✔	
CASC-22, Montreal	2009	✔	✔	✔
CASC-J5, Edinburgh	2010	✔	✔	✔
CASC-23, Wroclaw	2011	✔		✔
CASC-J6, Manchester	2012	✔		✔
total		12	11	4

Notes:

- In CASC-16 VAMPIRE came second after E-SETHEO, but E-SETHEO was retrospectively disqualified after the competition when one of its components E was found unsound.
- In 2000–2001 VAMPIRE used FLOTTER [35] implemented at MPI Informatik to transform formulas into clausal form; since 2002 clausal form transformation was handled by VAMPIRE itself.
- In 2010–2012 the clausifier of VAMPIRE was used by iProver, that won the EPR division of CASC.

- VAMPIRE can produce *local proofs* [13,19] in first-order logic with or without theories and extract interpolants [5] from them. Moreover, VAMPIRE can *minimise interpolants* using various measures, such as the total number of symbols or quantifiers [11].
- VAMPIRE has a special mode for working with *very large knowledge bases* and can *answer queries* to them.
- VAMPIRE can prove theorems in combinations of first-order logic and *theories*, such as integer arithmetic. It implements several *theory functions* on integers, real numbers, arrays, and strings. This makes VAMPIRE useful for reasoning with theories and quantifiers.
- VAMPIRE is fully compliant with the first-order part of the *TPTP syntax* [31,33] used by nearly all first-order theorem provers. It understands sorts and arithmetic. It was the first ever first-order theorem prover to implement the TPTP if-then-else and let-in formula and term constructors useful for program analysis.
- VAMPIRE supports several other *input syntaxes*, including the SMTLib syntax [2]. To perform program analysis, it also can read programs written in C.
- VAMPIRE can analyse *C programs with loops* and generate loop invariants using symbol elimination [18].
- VAMPIRE can produce *detailed proofs*. Moreover, it was the first ever theorem prover that produced proofs for *first-order* (as opposed to clausal form) derivations.
- VAMPIRE implements *many options* to help a user to control proof-search.
- VAMPIRE has a special *consequence elimination mode* that can be used to quickly remove from a set of formulas some formulas implied by other formulas in this set.
- VAMPIRE can run *several proof attempts in parallel* on a multi-core processor.
- VAMPIRE has a *liberal licence*, see [20].

Overview of the Paper

The rest of this paper is organised as follows. Sections 2-5 describe the underlining principles of first-order theorem proving and address various issues that are only implemented in VAMPIRE. Sections 6-11 present new and unconventional applications of first-order theorem proving implemented in VAMPIRE. Many features described below are implemented only in VAMPIRE.

- Section 2 (Mini-Tutorial) contains a brief tutorial explaining how to use VAMPIRE. This simple tutorial illustrates the most common use of VAMPIRE on an example, and is enough for inexperienced users to get started with it.
- To understand how VAMPIRE and other superposition theorem provers search for a proof, in Section 3 (Proof-Search by Saturation) we introduce the superposition inference system and the concept of saturation.
- To implement a superposition inference system one needs a saturation algorithm exploiting a powerful concept of *redundancy*. In Section 4 (Redundancy Elimination) we introduce this concept and explain how it is used in saturation algorithms. We also describe the three saturation algorithms used of VAMPIRE.
- The superposition inference system is operating on sets of clauses, that is formulas of a special form. If the input problem is given by arbitrary first-order formulas, the preprocessing steps of VAMPIRE are first applied as described in Section 5 (Preprocessing). Preprocessing is also applied to sets of clauses.
- VAMPIRE can be used in program analysis, in particular for loop invariant generation and interpolation. The common theme of these applications is the symbol elimination method. Section 6 (Coloured Proofs, Interpolation, and Symbol Elimination) explains symbol elimination and its use in VAMPIRE.
- In addition to standard first-order reasoning, VAMPIRE understands sorts, including built-in sorts integers, rationals, reals and arrays. The use of sorts and reasoning with theories in VAMPIRE is described in Section 7 (Sorts and Theories).
- In addition to checking unsatisfiability, VAMPIRE can also check satisfiability of a first-order formula using three different methods. These methods are overviewed in Section 8 (Satisfiability Checking Finding Finite Models).
- The proof-search, input, and output in VAMPIRE can be controlled by a number of options and modes. One of the most efficient theorem proving modes of VAMPIRE, called the CASC mode, uses the strategies used in last CASC competitions. VAMPIRE's options and the CASC mode are presented in Section 9 (VAMPIRE Options and the CASC Mode).
- VAMPIRE implements extensions of the TPTP syntax, including the TPTP if-then-else and let-in formula and term constructors useful for program analysis. The use of these constructs is presented in Section 10 (Advanced TPTP Syntax).
- Proving theorems is not the only way to use VAMPIRE. One can also use it for consequence finding, program analysis, linear arithmetic reasoning, clausification, grounding and some other purposes. These advanced features are described in Section 11 (Cookies).

```
%---- 1 * x = 1
fof(left_identity,axiom,
    ! [X] : mult(e,X) = X).
%---- i(x) * x = 1
fof(left_inverse,axiom,
    ! [X] : mult(inverse(X),X) = e).
%---- (x * y) * z = x * (y * z)
fof(associativity,axiom,
    ! [X,Y,Z] : mult(mult(X,Y),Z) = mult(X,mult(Y,Z))).
%---- x * x = 1
fof(group_of_order_2,hypothesis,
    ! [X] :  mult(X,X) = e).
%---- prove x * y = y * x
fof(commutativity,conjecture,
    ! [X] : mult(X,Y) = mult(Y,X)).
```

Fig. 1. A TPTP representation of a simple group theory problem

2 Mini-Tutorial

In this section we describe a simple way of using VAMPIRE for proving formulas. Using VAMPIRE is very easy. All one needs is to write the formula as a problem in the TPTP syntax [31,33] and run VAMPIRE on this problem. VAMPIRE is completely automatic. That is, once you started a proof attempt, it can only be interrupted by terminating VAMPIRE.

2.1 A Simple Example

Consider the following example from a group theory textbook: if all elements in a group have order 2, then the group is commutative. We can write down this problem in first-order logic using the language and the axioms of group theory, as follows:

$$\begin{array}{ll} & \forall x(1 \cdot x = x) \\ \text{Axioms (of group theory):} & \forall x(x^{-1} \cdot x = 1) \\ & \forall x \forall y \forall z((x \cdot y) \cdot z = x \cdot (y \cdot z)) \\ & \\ \text{Assumptions:} & \forall x(x \cdot x = 1) \\ \hline \text{Conjecture:} & \forall x \forall y(x \cdot y = y \cdot x) \end{array}$$

The problem stated above contains three axioms, one assumption, and a conjecture. The axioms above can be used in any group theory problem. However, the assumption and conjecture are specific to the example we study; they respectively express the order property (assumption) and the commutative property (conjecture).

The next step is to write this first-order problem in the TPTP syntax. TPTP is a Prolog-like syntax understood by all modern first-order theorem provers. A representation of our example in the TPTP syntax is shown in Figure 1. We should save this problem to a file, for example, `group.tptp`, and run VAMPIRE using the command::

first-order logic	TPTP
$\bot, \top$	`$false, $true`
$\neg F$	`~F`
$F_1 \wedge \ldots \wedge F_n$	`F1 & ... & Fn`
$F_1 \vee \ldots \vee F_n$	`F1 | ... | Fn`
$F_1 \rightarrow F_n$	`F1 => Fn`
$F_1 \leftrightarrow F_n$	`F1 <=> Fn`
$(\forall x_1) \ldots (\forall x_n) F$	`![X1,...,Xn]:F`
$(\exists x_1) \ldots (\exists x_n) F$	`?[X1,...,Xn]:F`

Fig. 2. Correspondence between the first-order logic and TPTP notations

```
vampire group.tptp
```

Let us consider the TPTP representation of Figure 1 in some detail. The TPTP syntax has *comments*. Any text beginning with the `%` symbol is considered a comment. For example, the line `%---- 1 * x = 1` is a comment. Comments are intended only as an additional information for human users and will be ignored by VAMPIRE.

The axiom $\forall x(1 \cdot x = x)$ appears in the input as `fof(left_identity, axiom, ![X] : mult(e,X) = X)`. The keyword `fof` means "first-order formula". One can use the keyword `tff` ("typed first-order formula") instead, see Section 7. VAMPIRE considers `fof` and `tff` as synonyms[1]. The word `left_identity` is chosen to denote the *name* of this axiom. The user can choose any other name. Names of input formulas are ignored by VAMPIRE when it searches for a proof but they can be used in the proof output.

The variable x is written as *capital* `X`. TPTP uses the Prolog convention for variables: variable names start with upper-case letters. This means that, for example, in the formula `mult(e,x) = x`, the symbol `x` will be a considered a constant.

The universal quantifier $\forall x$ is written as `![X]`. Note that the use of `![x] : mult(e,x) = x)` will result in a syntax error, since `x` is not a valid variable name. Unlike the Prolog syntax, the TPTP syntax does not allow one to use operators. Thus, one cannot write a more elegant `e * X` instead of `mult(e,X)`. The only exception is the equality (`=`) and the inequality symbols (`!=`), which must be written as operators, for example, in `mult(e,X) = X`. The correspondence between the first-order logic and TPTP notation is summarised in Figure 2.

2.2 Proof by Refutation

VAMPIRE tries to prove the conjecture of Figure 1 by adding the negation of the conjecture to the axioms and the assumptions and checking if the the resulting set of formulas is unsatisfiable. If it is, then the conjecture is a logical consequence of the axioms and the assumptions. A proof of unsatisfiability of a negation of formula is sometimes called

[1] Until recently, `fof(...)` syntax in TPTP was a special case of `tff(...)`. It seems that now there is a difference between the two syntaxes in the treatment of integers, which we hope will be removed in the next versions of the TPTP language.

```
Refutation found. Thanks to Tanya!
203. $false [subsumption resolution 202,14]
202. sP1(mult(sK,sK0)) [backward demodulation 188,15]
188. mult(X8,X9) = mult(X9,X8) [superposition 22,87]
87. mult(X2,mult(X1,X2)) = X1 [forward demodulation 71,27]
71. mult(inverse(X1),e) = mult(X2,mult(X1,X2)) [superposition 23,20]
27. mult(inverse(X2),e) = X2 [superposition 22,10]
23. mult(inverse(X4),mult(X4,X5)) = X5 [forward demodulation 18,9]
22. mult(X0,mult(X0,X1)) = X1 [forward demodulation 16,9]
20. e = mult(X0,mult(X1,mult(X0,X1))) [superposition 11,12]
18. mult(e,X5) = mult(inverse(X4),mult(X4,X5)) [superposition 11,10]
16. mult(e,X1) = mult(X0,mult(X0,X1)) [superposition 11,12]
15. sP1(mult(sK0,sK)) [inequality splitting 13,14]
14. ~sP1(mult(sK,sK0)) [inequality splitting name introduction]
13. mult(sK,sK0) != mult(sK0,sK) [cnf transformation 8]
12. e = mult(X0,X0) (0:5) [cnf transformation 4]
11. mult(mult(X0,X1),X2)=mult(X0,mult(X1,X2))[cnf transformation 3]
10. e = mult(inverse(X0),X0) [cnf transformation 2]
9. mult(e,X0) = X0 [cnf transformation 1]
8. mult(sK,sK0) != mult(sK0,sK) [skolemisation 7]
7. ? [X0,X1] : mult(X0,X1) != mult(X1,X0) [ennf transformation 6]
6. ~! [X0,X1] : mult(X0,X1) = mult(X1,X0) [negated conjecture 5]
5. ! [X0,X1] : mult(X0,X1) = mult(X1,X0) [input]
4. ! [X0] : e = mult(X0,X0)[input]
3. ! [X0,X1,X2] : mult(mult(X0,X1),X2) = mult(X0,mult(X1,X2)) [input]
2. ! [X0] : e = mult(inverse(X0),X0) [input]
1. ! [X0] : mult(e,X0) = X0 [input]
```

Fig. 3. VAMPIRE's Refutation

a *refutation* of this formula, so such proofs are commonly referred to as proofs by refutation.

Figure 3 shows a (slightly modified) refutation found by VAMPIRE for our simple group theory example. Let us analyse this refutation, since it contains many concepts discussed further in this paper.

Proof outputs by VAMPIRE are dags, whose nodes are labelled by formulas. Every formula is assigned a unique number, in our example they are numbered `1` to `203`. Numbers are assigned to formulas during the proof search, in the order in which they are generated. The proof consists of *inferences*. Each inference infers a formula, called the *conclusion* of this inference, from a set of formulas, called the *premises* of the inference. For example, formula `87` is inferred from formulas `71` and `27`. We can also say that formulas `71` and `27` are *parents* of formula `87`. Some formulas (including formulas read from the input file) have no parents. In this example these are formulas `1`–`5`. VAMPIRE sometimes packs long chains of proof steps into a single inference, resulting in clauses with many parents, sometimes over a hundred. The dag proof of Figure 3 is rooted at formula `203`: `$false`. To see that the proof is not a tree consider, for example, formula `11`: it is used to infer three different formulas (`16`, `18` and `20`).

Each formula (or inference) in the proof is obtained using one or more *inference rules*. They are shown in brackets, together with parent numbers. Some examples of inference rules in this proof are `superposition`, `inequality splitting` and `skolemisation`. All together, VAMPIRE implements 79 inference rules.

There are several kinds of inference rules. Some inferences, marked as `input`, introduce input formulas. There are many inference rules related to preprocessing input

formulas, for example `ennf transformation` and `cnf transformation`. Preprocessing is discussion in Section 5. The input formulas are finally converted to *clauses*, after which VAMPIRE tries to check unsatisfiability of the resulting set of clauses using the *resolution and superposition inference system* discussed in Section 3. The superposition calculus rules can generally be divided in two parts: *generating* and *simplifying* ones. This distinction will be made more clear when we later discuss *saturation* and *redundancy elimination* in Section 4. Though the most complex part of proof search is the use of the resolution and superposition inference system, preprocessing is also very important, especially when the input contains deep formulas or a large number of formulas.

Formulas and clauses having free variables are considered implicitly universally quantified. Normally, the conclusion of an inference is a logical consequence of its premises, though in general this is not the case. Notable exception are inference rules that introduce new symbols: in our example these are `skolemisation` and `inequality splitting`. Though not preserving logical consequence, inference rules used by VAMPIRE guarantee soundness, which means that an inference cannot change a satisfiable set of formulas into an unsatisfiable one.

One can see that the proof of Figure 3 is a refutation: the top line explicitly mentions that a refutation is found, the proof derives the false formula `$false` in inference `203` and inference `6` negates the input conjecture.

Finally, the proof output of VAMPIRE contains only a subset of all generated formulas. The refutation of Figure 3 contains 26 formulas, while 203 formulas were generated. It is not unusual that very short proofs require many generated formulas and require a lot of time to find.

Besides the refutation, the output produced by VAMPIRE contains *statistics* about the proof attempt. These statistics include the overall running time, used memory, and the termination reason (for example, `refutation found`). In addition, it contains information about the number of various kinds of clause and inferences. These statistics are printed even when no refutation is found. An example is shown in Figure 4.

If one is only interested in provability but not in proofs, one can disable the refutation output by using the option `-- proof off`, though normally this does not result in considerable improvements in either time or memory.

In some cases, VAMPIRE might report `Satisfiability detected`. Remember that VAMPIRE tries to find a *refutation*: given a conjecture F, he tries to establish (un)satisfiability of its negation $\neg F$. Hence, when a conjecture$F$ is given, "Satisfiability" refers not to the satisfiability of F but to the satisfiability of $\neg F$.

2.3 Using Time and Memory Limit Options

For very hard problems it is possible that VAMPIRE will run for a very long time without finding a refutation or detecting satisfiability. As a rule of thumb, one should *always run Vampire with a time limit*. To this end, one should use the parameter `-t` or `--time_limit`, specifying how much time (in seconds) it is allowed to spend for proof search. For example, `vampire -t 5 group.tptp` calls VAMPIRE on the input file `group.tptp` with the time limit of 5 seconds. If a refutation cannot be

```
Version: Vampire 2.6 (revision 1692)
Termination reason: Refutation

Active clauses: 14
Passive clauses: 35
Generated clauses: 194
Final active clauses: 8
Final passive clauses: 11
Input formulas: 5
Initial clauses: 6

Split inequalities: 1

Fw subsumption resolutions: 1
Fw demodulations: 68
Bw demodulations: 14

Forward subsumptions: 65
Backward subsumptions: 1
Fw demodulations to eq. taut.: 20
Bw demodulations to eq. taut.: 1

Forward superposition: 60
Backward superposition: 39
Self superposition: 6

Unique components: 6

Memory used [KB]: 255
Time elapsed: 0.007 s
```

Fig. 4. Statistics output by VAMPIRE

found within the given time limit, VAMPIRE specifies `time limit expired` as the termination reason.

VAMPIRE tries to adjust the proof search pace to the time limit which might lead to strange (but generally pleasant) effects. To illustrate it, suppose that VAMPIRE without a time limit finds a refutation of a problem in 10 seconds. This does not mean that with a time limit of 9 seconds VAMPIRE will be unable to find a refutation. In fact, in most case it will be able to find it. We have examples when it can find proofs when the time limit is set to less than 5% of the time it needs to find a proof without the time limit. A detailed explanation of the magic used can be found in [27].

One can also specify the *memory limit* (in Megabytes) of VAMPIRE by using the parameter `-m` or `--memory_limit`. If VAMPIRE does not have enough memory, it will not give up immediately. Instead, it will try to discard some clauses from the search space and reuse the released memory.

2.4 Limitations

What if VAMPIRE can neither find a refutation nor establish satisfiability for a given time limit? Of course, one can increase the time limit and try again. Theoretically, VAMPIRE is based on a *complete inference system*, that is, if the problem is

unsatisfiable, then given enough time and space VAMPIRE will eventually find a refutation. In practice, theorem proving in first-order logic is a very hard problem, so it is unreasonable to expect provers to find a refutation quickly or to find it at all. When VAMPIRE cannot find a refutation, increasing time limit is not the only option to try. VAMPIRE has many other parameters controlling search for a refutation, some of them are mentioned in later sections of this paper.

3 Proof-Search by Saturation

Given a problem, VAMPIRE works as follows:

- *read the problem*;
- determine *proof-search options* to be used for this problem;
- *preprocess the problem*;
- *convert it into conjunctive normal form (cnf)*;
- run a *saturation algorithm* on it;
- report the *result*, maybe including a refutation.

In this section we explain the concept of saturation-based theorem proving and present a simple saturation algorithm. The other parts of the inference process are described in the following sections.

Saturation is the underlying concept behind the proof-search algorithms using the resolution and superposition inference system. In the sequel we will refer to theorem proving using variants of this inference system as simply *superposition theorem proving*. This section explains the main concepts and ideas involved in superposition theorem proving and saturation. A detailed exposition of the theory of superposition can be found in [1,26]. This section is more technical than the other sections of the paper: we decided to present superposition and saturation in more details, since the concept of saturation is relatively unknown outside of the theorem proving community. Similar algorithms are used in some other areas, but they are not common. For example, Buchberger's algorithm for computing Gröbner basis [4] can be considered as an example of a saturation algorithm.

Given a set S of formulas and an *inference system* $\mathbb{I}$, one can try to *saturate* this set with respect to the inference system, that is, to build a set of formulas that contains S and is closed under inferences in $\mathbb{I}$. Superposition theorem provers perform inferences on formulas of a special form, called *clauses*.

3.1 Basic Notions

We start with an overview of relevant definitions and properties of first-order logic, and fix our notation. We consider the standard first-order predicate logic with equality. We allow all standard boolean connectives and quantifiers in the language. We assume that the language contains the logical constants $\top$ for always true and $\bot$ for always false formulas.

Throughout this paper, we denote terms by l, r, s, t, variables by x, y, z, constants by a, b, c, d, e, function symbols by f, g, h, and predicate symbols by p, q. As usual, an

atom is a formula of the form $p(t_1, \ldots, t_n)$, where p is a predicate symbol and $t_1, \ldots, t_n$ are terms. The equality predicate symbol is denoted by $=$. Any atom of the form $s = t$ is called an *equality*. By $s \neq t$ we denote the formula $\neg(s = t)$.

A *literal* is an atom A or its negation $\neg A$. Literals that are atoms are called *positive*, while literals of the form $\neg A$ *negative*. A *clause* is a disjunction of literals $L_1 \vee \ldots \vee L_n$, where $n \geq 0$. When $n = 0$, we will speak of the *empty clause*, denoted by $\Box$. The empty clause is always false. If a clause contains a single literal, that is, $n = 1$, it is called a *unit clause*. A unit clause with $=$ is called an *equality literal*. We denote atoms by A, literals by L, clauses by C, D, and formulas by F, G, R, B, possibly with indices.

Let F be a formula with free variables $\bar{x}$, then $\forall F$ (respectively, $\exists F$) denotes the formula $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$). A formula is called *closed*, or a *sentence*, if it has no free variables. We call a *symbol* a predicate symbol, a function symbol or a constant. Thus, variables are not symbols. We consider equality $=$ part of the language, that is, equality is not a symbol. A formula or a term is called *ground* if it has no occurrences of variables. A formula is called *universal* (respectively, *existential)* if it has the form $(\forall \bar{x})F$ (respectively, $(\exists \bar{x})F$), where F is quantifier-free. We write $C_1, \ldots, C_n \vdash C$ to denote that the formula $C_1 \wedge \ldots \wedge C_n \rightarrow C$ is a tautology. Note that $C_1, \ldots, C_n, C$ may contain free variables.

A *signature* is any finite set of symbols. The *signature of a formula* F is the set of all symbols occurring in this formula. For example, the signature of $b = g(z)$ is $\{g, b\}$. The *language of a formula* F is the set of all formulas built from the symbols occurring in F, that is formulas whose signatures are subsets of the signature of F.

We call a *theory* any set of closed formulas. If T is a theory, we write $C_1, \ldots, C_n \vdash_T C$ to denote that the formula $C_1 \wedge \ldots \wedge C_1 \rightarrow C$ holds in all models of T. In fact, our notion of theory corresponds to the notion of *axiomatisable theory* in logic. When we work with a theory T, we call symbols occurring in T *interpreted* while all other symbols *uninterpreted*.

We call a *substitution* any expression θ of the form $\{x_1 \mapsto t_1, \ldots, x_n \mapsto t_n\}$, where $n \geq 0$. An *application* of this substitution to an expression E, denoted by $E\theta$, is the expression obtained from E by the simultaneous replacements of each x_i by t_i. By an expression here we mean a term, atom, literal, or clause. An expression is *ground* if it contains no variables.

We write $E[s]$ to mean an expression E with a particular occurrence of a term s. When we use the notation $E[s]$ and then write $E[t]$, the latter means the expression obtained from $E[s]$ by replacing the distinguished occurrence of s by the term t.

A *unifier* of two expressions E_1 and E_2 is a substitution θ such that $E_1\theta = E_2\theta$. It is known that if two expressions have a unifier, then they have a so-called *most general unifier (mgu)*. For example, consider terms $f(x_1, g(x_1), x_2)$ and $f(y_1, y_2, y_2)$. Some of their unifiers are $\theta_1 = \{y_1 \mapsto x_1, y_2 \mapsto g(x_1), x_2 \mapsto g(x_1)\}$ and $\theta_2 = \{y_1 \mapsto a, y_2 \mapsto g(a), x_2 \mapsto g(a), x_1 \mapsto a\}$, but only θ_1 is most general. There are several algorithms for finding most general unifiers, from linear or almost linear [23] to exponential [28] ones, see [12] for an overview. In a way, VAMPIRE uses none of them since all unification-related operations are implemented using *term indexing* [30].

3.2 Inference Systems and Proofs

An *inference rule* is an n-ary relation on formulas, where $n \geq 0$. The elements of such a relation are called *inferences* and usually written as:

$$\frac{F_1 \quad \ldots \quad F_n}{F} .$$

The formulas $F_1, \ldots, F_n$ are called the *premises* of this inference, whereas the formula F is the *conclusion* of the inference. An *inference system* $\mathbb{I}$ is a set of inference rules. An *axiom* of an inference system is any conclusion of an inference with 0 premises. Any inferences with 0 premises and a conclusion F will be written without the bar line, simply as F.

A *derivation* in an inference system $\mathbb{I}$ is a tree built from inferences in $\mathbb{I}$. If the root of this derivation is F, then we say it is a *derivation of* F. A derivation of F is called a *proof* of F if it is finite and all leaves in the derivation are axioms. A formula F is called *provable* in $\mathbb{I}$ if it has a proof. We say that a derivation of F is *from assumptions* $F_1, \ldots, F_m$ if the derivation is finite and every leaf in it is either an axiom or one of the formulas $F_1, \ldots, F_m$. A formula F is said to be *derivable from* assumptions $F_1, \ldots, F_m$ if there exists a derivation of F from $F_1, \ldots, F_m$. A *refutation* is a derivation of $\bot$.

Note that a proof is a derivation from the empty set of assumptions. Any derivation from a set of assumptions S can be considered as a derivation from any larger set of assumptions $S' \supseteq S$.

3.3 A Simple Inference System and Completeness

We give now a simple inference system for first-order logic with equality, called the *superposition inference system*. For doing so, we first introduce the notion of a *simplification ordering* on terms, as follows. An ordering $\succ$ on terms is called a *simplification ordering* if it satisfies the following conditions:

1. $\succ$ is *well-founded*, that is there exists no infinite sequence of terms $t_0, t_1, \ldots$ such that $t_0 \succ t_1 \succ \ldots$.
2. $\succ$ is *monotonic*: if $l \succ r$, then $s[l] \succ s[r]$ for all terms s, l, r.
3. $\succ$ is *stable under substitutions*: if $l \succ r$, then $l\theta \succ r\theta$.
4. $\succ$ has the *subterm property*: if r is a subterm of l and $l \neq r$, then $l \succ r$.

Given a simplification ordering on terms, we can extend it to a simplification ordering on atoms, literals, and even clauses. For details, see [26,1,7]. One of the important things to know about simplification orderings is that they formalise a notion of "being simpler" on expressions. For example, for the Knuth-Bendix ordering [14], if a ground term s has fewer symbols than a ground term t, then $t \succ s$.

In addition, to simplification orderings, we need a concept of a *selection function*. A selection function selects in every non-empty clause a non-empty subset of literals. When we deal with a selection function, we will underline selected literals: if we write a clause in the form $\underline{L} \vee C$, it means that L (and maybe some other literals) are selected

in $L \vee C$. One example of a selection function sometimes used in superposition theorem provers is the function that selects all maximal literals with respect to the simplification ordering used in the system.

The superposition inference system is, in fact, a family of systems, parametrised by a simplification ordering and a selection function. We assume that a simplification ordering and a selection function are fixed and will now define the *superposition inference system*. This inference system, denoted by Sup, consists of the following rules:

Resolution.

$$\frac{\underline{A} \vee C_1 \quad \underline{\neg A'} \vee C_2}{(C_1 \vee C_2)\theta} \ ,$$

where θ is a mgu or A and A'.

Factoring.

$$\frac{\underline{A} \vee \underline{A'} \vee C}{(A \vee C)\theta} \ ,$$

where θ is a mgu or A and A'.

Superposition.

$$\frac{\underline{l = r} \vee C_1 \quad \underline{L[s]} \vee C_2}{(L[r] \vee C_1 \vee C_2)\theta} \qquad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] = t'} \vee C_2}{(t[r] = t' \vee C_1 \vee C_2)\theta} \qquad \frac{\underline{l = r} \vee C_1 \quad \underline{t[s] \neq t'} \vee C_2}{(t[r] \neq t' \vee C_1 \vee C_2)\theta} \ ,$$

where θ is a mgu of l and s, s is not a variable, $r\theta \not\succeq l\theta$, (first rule only) $L[s]$ is not an equality literal, and (second and third rules only) $t'\theta \not\succeq t[s]\theta$.

Equality Resolution.

$$\frac{\underline{s \neq t} \vee C}{C\theta} \ ,$$

where θ is a mgu of s and t.

Equality Factoring.

$$\frac{\underline{s = t} \vee \underline{s' = t'} \vee C}{(s = t \vee t \neq t' \vee C)\theta} \ ,$$

where θ is an mgu of s and s', $t\theta \not\succeq s\theta$, and $t'\theta \not\succeq t\theta$.

VAMPIRE uses the names of inference rules in its proofs and statistics. For example, the proof of Figure 3 and statistics displayed in Figure 4 use `superposition`. The term `demodulation` used in them also refers to superposition, as we shall see later.

If the selection function is *well-behaved*, that is, it either selects a negative literal or all maximal literals, then the superposition inference system is both *sound* and *refutationally complete*. By soundness we mean that, if the empty clause $\Box$ is derivable from a set S of formulas in Sup, then S is unsatisfiable. By (refutational) completeness we mean that if a set S of formulas is unsatisfiable, then $\Box$ is derivable from S in Sup.

Defining a sound and complete inference system is however not enough for automatic theorem proving. If we want to use the sound and complete inference system of Sup for finding a refutation, we have to understand how to *organise the search for a refutation in* Sup. One can apply *all possible inferences* to clauses in the search space in a certain order until we derive the empty clause. However, a simple implementation of this idea will hardly result in an efficient theorem prover, because blind applications

of all possible inferences will blow up the search space very quickly. Nonetheless, the idea of generating all clauses derivable from S is the key idea of saturation-based theorem proving and can be made very efficient when one exploits a powerful concept of *redundancy* and uses good *saturation algorithms*.

3.4 Saturation

A set of clauses S is called *saturated with respect to an inference system* $\mathbb{I}$ if, for every inference in $\mathbb{I}$ with premises in S, the conclusion of this inference belongs to S too. When the inference system is clear from the context, in this paper it is always Sup, we simply say "saturated set of clauses". It is clear that for every set of clauses S there exists the smallest saturated set containing S: this set consists of all clauses derivable from S.

From the completeness of Sup we can then conclude the following important property. A set S of clauses is unsatisfiable if and only if the smallest set of clauses containing S and saturated with respect to Sup also contains the empty clause.

To saturate a set of clauses S with respect to an inference system, in particular Sup, we need a *saturation algorithm*. At every step such an algorithm should select an inference, apply this inference to S, and add conclusions of the inferences to the set S. If at some moment the empty clause is obtained, we can conclude that the input set of clauses is unsatisfiable. A good strategy for *inference selection* is crucial for an efficient behaviour of a saturation algorithm. If we want a saturation algorithm to preserve completeness, that is, to guarantee that a saturated set is eventually built, the inference selection strategy must be *fair*: every possible inference must be selected at some step of the algorithm. A saturation algorithm with a fair inference selection strategy is called a *fair saturation algorithm*.

By completeness of Sup, there are three possible scenarios for running a fair saturation algorithm on an input set of clauses S:

1. At some moment the empty clause $\square$ is generated, in this case S is unsatisfiable.
2. Saturation terminates without ever generating $\square$, in this case S is satisfiable.
3. Saturation runs forever, but without generating $\square$. In this case S is satisfiable.

Note that in the third case we do not establish satisfiability of S after a finite amount of time. In reality, in this case, a saturation-based prover will simply run out of resources, that is terminate by time or memory limits, or will be interrupted. Even when a set of clauses is unsatisfiable, termination by a time or memory limit is not unusual. Therefore in practice the third possibility must be replaced by:

3'. Saturation will run *until the system runs out of resources,* but without generating $\square$. In this case it is unknown whether S is unsatisfiable.

4 Redundancy Elimination

However, a straightforward implementation of a fair saturation algorithm will not result in an efficient prover. Such an implementation will not solve some problems rather trivial for modern provers because of the rapid growth of the search space. This is due to two reasons:

1. the superposition inference system has many inferences that can be avoided;
2. some clauses can be removed from the search space without compromising completeness.

In other words,

1. some *inferences* in the superposition system are *redundant;*
2. some *clauses* in the search space are *redundant.*

To have an efficient prover, one needs to exploit a powerful concept of *redundancy* and *saturation up to redundancy*. This section explains the concept of redundancy and how it influences the design and implementation of saturation algorithms.

The modern theory of resolution and superposition [1,26] deals with inference systems in which clauses can be *deleted* from the search space. Remember that we have a simplification ordering $\succ$, which can also be extended to clauses. There is a general redundancy criterion: given a set of clauses S and a clause $C \in S$, C is *redundant* in S if it is a logical consequence of those clauses in S that are strictly smaller than C w.r.t. $\succ$. However, this general redundancy criterion is undecidable, so theorem provers use some sufficient conditions to recognise redundant clauses. Several specific redundancy criteria based on various sufficient conditions will be defined below.

Tautology Deletion. A clause is called a *tautology* if it is a valid formula. Examples of tautologies are clauses of the form $A \vee \neg A \vee C$ and $s = s \vee C$. Since tautologies are implied by the empty set of formulas, they are redundant in every clause set. There are more complex equational tautologies, for example, $a \neq b \vee b \neq c \vee a = c$. Equational tautology checking can be implemented using congruence closure. It is implemented in VAMPIRE and the number of removed tautologies appears in the statistics.

Subsumption. We say that a clause C subsumes a clause D if D can be obtained from C by two operations: application of a substitution θ and adding zero or more literals. In other words, $C\theta$ is a submultiset of D if we consider clauses as multisets of their literals. For example, the clause $C = p(a, x) \vee r(x, b)$ subsumes the clause $D = r(f(y), b) \vee q(y) \vee p(a, f(y))$, since D can be obtained from C by applying the substitution $\{x \mapsto f(y)\}$ and adding the literal $q(y)$. Subsumed clauses are redundant in the following sense: if a clause set S contains two different clauses C and D and C subsumes D, then D is redundant in S. Although subsumption checking is NP-complete, it is a powerful redundancy criterion. It is implemented in VAMPIRE and its use is controlled by options.

The concept of redundancy allows one to remove clauses from the search space. Therefore, an inference process using this concept consists of steps of two kinds:

1. add to the search space a new clause obtained by an inference whose premises belong to the search space;
2. delete a redundant clause from the search space.

Before defining saturation algorithms that exploit the concept of redundancy, we have to define a new notion of inference process and reconsider the notion of fairness. Indeed, for this kind of process we cannot formulate fairness in the same way as before, since an inference enabled at some point of the inference process may be disabled afterwards, if one or more of its parents are deleted as redundant.

To formalise an inference process with clause deletion, we will consider such a process as a sequence $S_0, S_1, \ldots$ of sets of clauses. Intuitively, S_0 is the initial set of clauses and S_i for $i \geq 0$ is the search space at the step i of the process. An *inference process* is any (finite or infinite) sequence of sets of formulas $S_0, S_1, \ldots$, denoted by:

$$S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots \tag{1}$$

A *step* of this process is a pair $S_i \Rightarrow S_{i+1}$.

Let $\mathbb{I}$ be an inference system, for example the superposition inference system Sup. An inference process is called an $\mathbb{I}$-process if each of its steps $S_i \Rightarrow S_{i+1}$ has one of the following two properties:

1. $S_{i+1} = S_i \cup \{C\}$ and $\mathbb{I}$ contains an inference
$$\frac{C_1 \quad \ldots \quad C_n}{C}$$
such that $\{C_1, \ldots, C_n\} \subseteq S_i$.
2. $S_{i+1} = S_i - \{C\}$ such that C is redundant in S_i.

In other words, every step of an $\mathbb{I}$-derivation process either adds to the search space a conclusion of an $\mathbb{I}$-inference or deletes from it a redundant clause.

An inference process can delete clauses from the search space. To define fairness we are only interested in clauses that are never deleted. Such clauses are called *persistent*. Formally, a clause C is *persistent* in an inference process (1) if for some step i it belongs to all sets S_j for which $j \geq i$. In other words, a persistent clause occurs in S_i and is not deleted at steps $S_i \Rightarrow S_{i+i} \Rightarrow \ldots$. An inference process (1) is called *fair* if it satisfies the following principle: every possible inference with persistent clauses as premises must be performed at some step.

The superposition inference system Sup has a very strong completeness property formulated below.

THEOREM 1 (COMPLETENESS). Let Sup be the superposition inference system, S_0 be a set of clauses and $S_0 \Rightarrow S_1 \Rightarrow S_2 \Rightarrow \ldots$ be a fair Sup-inference process. Then S_0 is unsatisfiable if and only if some S_i contains the empty clause.

An algorithm of *saturation up to redundancy* is any algorithm that implements inference processes. Naturally, we are interested in *fair saturation algorithms* that guarantee fair behaviour for every initial set of clauses S_0.

4.1 Generating and Simplifying Inferences

Deletion of redundant clauses is desirable since every deletion reduces the search space. If a newly generated clause makes some clauses in the search space redundant, adding

such a clause to the search space comes "at no cost", since it will be followed by deletion of other (more complex) clauses. This observation gives rise to an idea of prioritising inferences that make one or more clauses in the search space redundant. Since the general redundancy criterion is undecidable, we cannot in advance say whether an inference will result in a deletion. However, one can try to find "cheap" sufficient conditions for an inference to result in a deletion and try to search for such inferences in an eager way. This is exactly what the modern theorem provers do.

Simplifying Inferences. Let S be an inference of the form

$$\frac{C_1 \quad \dots \quad C_n}{C} \ .$$

We call this inference *simplifying* if at least one of the premises C_i becomes redundant after the addition of the conclusion C to the search space. In this case we also say that we *simplify* the clause C_i into C since this inference can be implemented as the replacement of C_i by C. The reason for the name "simplifying" is due to the fact that C is usually "simpler" than C_i in some strict mathematical sense. In a way, such inferences simplify the search space by replacing clauses by simpler ones. Let us consider below two examples of a simplifying rule. In these examples, and all further examples, we will denote the deleted clause by drawing a line through it, for example ~~$B \vee D$~~.
Subsumption resolution is one of the following inference rules:

$$\frac{A \vee C \quad \neg B \vee D}{D} \quad \text{or} \quad \frac{\neg A \vee C \quad B \vee D}{D} \ ,$$

such that for some substitution θ we have $A\theta \vee C\theta \subseteq B \vee D$, where clauses are considered as sets of literals. The name "subsumption resolution" is due to the fact that the applicability of this inference can be checked in a way similar to subsumption checking.
Demodulation is the following inference rule:

$$\frac{l = r \quad C[l\theta]}{C[r\theta]}$$

where $l\theta \succ r\theta$ and $(l = r)\theta \succ C[l\theta]$. Demodulation is also sometimes called *rewriting by unit equalities*. One can see that demodulation is a special case of superposition where one of the parents is deleted. The difference is that, unlike superposition, demodulation does not have to be applied only to selected literals or only into the larger part of equalities.

Generating Inferences. Inferences that are not simplifying are called *generating*: instead of simplifying one of the clauses in the search space, they generate a new clause C.

Many theorem provers, including VAMPIRE, implement the following principle:

apply simplifying inferences eagerly; apply generating inferences lazily.

This principle influences the design of saturation algorithms in the following way: from time to time provers try to search for simplifying inferences at the expense of delaying generating inferences. More precisely, after generating each new clause C, VAMPIRE tries to apply as many simplifying rules using C as possible. It is often the case that a single simplifying inference gives rise to new simplifying inferences, thus producing long chains of them after a single generating inference.

Deletion Rules. Even when simplification rules are in use, deletion of redundant clauses is still useful and performed by VAMPIRE. VAMPIRE has a collection of *deletion rules*, which check whether clauses are redundant due to the presence of other clauses in the search space. Typical deletion rules are subsumption and tautology deletion. Normally, these deletion rules are applied to check if a newly generated clause is redundant (forward deletion rules) and then to check if one of the clauses in the search space becomes redundant after the addition of the new clause to the search space (backward deletion rules). An example of a forward deletion rule is forward demodulation, listed also in the proof output of Figure 3.

4.2 A Simple Saturation Algorithm

We will now present a *simple algorithm for saturation up to redundancy*. This algorithm is not exactly the saturation algorithm of VAMPIRE but it is a good approximation and will help us to identify various parts of the inference process used by VAMPIRE. The algorithm is given in Figure 5. In this algorithm we mark by ✓ parts that are informally explained in the rest of this section.

The algorithm uses three variables: a set $kept$ of so called *kept clauses*, a set *unprocessed* for storing clauses to be processed, and a clause new to represent the currently processed clause. The set $unprocessed$ is needed to make the algorithm search for simplification eagerly: as soon as a clause has been simplified, it is added to the set of unprocessed clauses so that we could check whether it can be simplified further or simplify other clauses in the search space. Moreover, the algorithm is organised in such a way that $kept$ is always *inter-reduced*, that is, all possible simplification rules between clauses in $kept$ have already been applied.

Let us now briefly explain parts of the saturation algorithm marked by ✓.

Retention Test. This is the test to decide whether the new clause should be kept or discarded. The retention test applies deletion rules, such as subsumption, to the new clause. In addition, VAMPIRE has other criteria to decide whether a new clause should be kept, for example, it can decide to discard too big clauses, even when this compromises completeness. VAMPIRE has several parameters to specify which deletion rules to apply and which clauses should be discarded. Note that in the last line the algorithm returns either *satisfiable* or *unknown*. It returns *satisfiable* when only redundant clauses were discarded by the retention test. When at least one non-redundant clause was discarded, the algorithm returns *unknown*.

```
   var kept, unprocessed: sets of clauses;
   var new: clause;
   unprocessed := the initial sets of clauses;
   kept := ∅;
   loop
     while unprocessed ≠ ∅
       new := select(unprocessed);
       if new = □ then return unsatisfiable;
✓      if retained(new) then                                   (* retention test *)
✓        simplify new by clauses in kept ;                     (* forward simplification *)
         if new = □ then return unsatisfiable;
✓        if retained(new) then                                 (* another retention test *)
✓          delete and simplify clauses in kept using new;      (* backward simplification *)
           move the simplified clauses from kept to unprocessed;
           add new to kept
     if there exists an inference with premises in kept not selected previously then
✓      select such an inference;                               (* inference selection *)
✓      add to unprocessed the conclusion of this inference     (* generating inference *)
     else return satisfiable or unknown
```

Fig. 5. A Simple Saturation Algorithm

Forward Simplification. In this part of the saturation algorithm, VAMPIRE tries to simplify the new clause by kept clauses. Note that the simplified clause can again become subject to deletion rules, for example it can become simplified to a tautology. For this reason after forward simplification another retention test may be required.

Backward Simplification. In this part, VAMPIRE tries to delete or simplify kept clauses by the new clause. If some clauses are simplified, they become candidates for further simplifications and are moved to *unprocessed*.

Inference Selection. This part of the saturation algorithm selects an inference that has not previously been selected. Ideally, inference selection should be fair. VAMPIRE

Generating Inference. This part of the saturation algorithm applies the selected inference. The inference generates a new clause, so we call it a generating inference.[2] Most of the generating inferences to be used are determined by VAMPIRE but some are user-controlled.

4.3 Saturation Algorithms of VAMPIRE

To design a saturation algorithm, one has to understand how to achieve fairness and how to organise redundancy elimination. VAMPIRE implements three different saturation algorithms. A description and comparison of these algorithms can be found in [27].

[2] This inference may turn out to be simplifying inference, since the new clause generated by it may sometimes simplify a kept clause.

A saturation algorithm in VAMPIRE can be chosen by setting the `--saturation_algorithm` option to one of the values `lrs`, `discount`, and `otter`. The `lrs` saturation algorithm is the default one used by VAMPIRE and refers to the *limited resource strategy algorithm*, `discount` is the *Discount algorithm*, and `otter` is the *Otter saturation algorithm*. In this section we briefly describe these three saturation algorithms.

Given Clause Algorithms. All saturation algorithms implemented in VAMPIRE belong to the family of *given clause algorithms*. These algorithms achieve fairness by implementing inference selection using *clause selection*. At each iteration of the algorithm a clause is selected and all generating inferences are performed between this clause and previously selected clauses. The currently selected clause is called the *given clause* in the terminology of [24].

Clause Selection. Clause selection in VAMPIRE is based on two parameters: the *age* and the *weight* of a clause. VAMPIRE maintains two priority queues, in which older and lighter clauses are respectively prioritised. The clauses are selected from one of the queues using an *age-weight ratio*, that is, a pair of non-negative integers (a, w). If the age-weight ratio is (a, w), then of each $a + w$ selected clauses a clauses will be selected from the age priority queue and w from the weight priority queue. In other words, of each $a + w$ clauses, a oldest and w lightest clauses are selected. The age-weight ratio can be controlled by the parameter `--age_weight_ratio`. For example, `--age_weight_ratio 2:3` means that of each 5 clauses, 2 will be selected by age and 3 by weight.

The *age* is implemented using numbering of clauses. Each kept clause is assigned a unique number. The numbers are assigned in the increasing order, so older clauses have smaller numbers. Each clause is also assigned a *weight*. By default, the weight of a clause is equal to its size, that is, the count of symbols in it, but it can also be controlled by the user, for example, by using the option `--nongoal_weight_coefficient`.

Active and Passive Clauses. Given clause algorithms distinguish between kept clauses previously selected for inferences and those not previously selected. Only the former clauses participate in generating inferences. For this reason they are called *active*. The kept clauses still waiting to be selected are called *passive*. The interesting question is whether passive clauses should participate in simplifying inferences.

Otter Saturation Algorithm. The first family of saturation algorithms do use passive clauses for simplifications. Any such saturation algorithm is called an *Otter* saturation algorithm after the theorem prover Otter [24]. The Otter saturation algorithm was designed as a result of research in resolution theorem proving in Argonne National Laboratory, see [22] for an overview.

Retention Test. The *retention test* in VAMPIRE mainly consists of deletion rules plus the weight test. The weight test discards clauses whose weight exceeds certain limit. By default, there is no limit on the weight. The weight limit can be controlled by using the option `--max_weight`. In some cases VAMPIRE may change the weight limit. This happens when it is running out of memory or when the limited resource strategy is on. Note that the weight test can discard a non-redundant clause. So when the weight

limit was set for at least some time during the proof-search, VAMPIRE will never return "satisfiable".

The growth of the number of kept clauses in the Otter algorithm causes fast deterioration of the processing rate of active clauses. Thus, when a complete procedure based on the Otter algorithm is used, even passive clauses with high selection priority often have to wait indefinitely long before they become selected. In theorem provers based on the Otter algorithm, all solutions to the completeness-versus-efficiency problem are based on the same idea: some non-redundant clauses are discarded from the search space.

Limited Resource Strategy Algorithm. This variation of the Otter saturation algorithm was introduced in one of the early versions of VAMPIRE. It is described in detail in [27]. This strategy is based on the Otter saturation algorithm and can be used only when a time limit is set.

The main idea of the limited resource strategy is the following. The system tries to identify which passive and unprocessed clauses have no chance to be selected by the time limit and discards these clauses. Such clauses will be called *unreachable*. Note that unreachability is fundamentally different from redundancy: redundant clauses are discarded without compromising completeness, while the notion of an unreachable clause makes sense only in the context of reasoning with a time limit.

To identify unreachable clauses, VAMPIRE measures the time spent by processing a given clause. Note that usually this time increases because the sets of active and passive clauses are growing, so generating and simplifying inferences are taking increasingly longer time. From time to time VAMPIRE estimates how the proof search pace will develop towards the time limit and, based on this estimation, identifies potentially unreachable clauses. Limited resource strategy is implemented by adaptive weight limit changes, see [27] for details. It is very powerful and is generally the best saturation algorithm in VAMPIRE.

Discount Algorithm. Typically, the number of passive clauses is much larger than the number of active ones. It is not unusual that the number of active clauses is less than 1% of the number of passive ones. As a result, inference pace may become dominated by simplification operations on passive clauses. To solve this problem, one can make passive clauses truly passive by not using them for simplifying inferences. This strategy was first implemented in the theorem prover Discount [6] and is called the Discount algorithm.

Since only a small subset of all clauses is involved in simplifying inferences, processing a new clause is much faster. However, this comes at a price. Simplifying inferences between a passive and a new clause performed by the Otter algorithm may result in a valuable clause, which will not be found by the Discount algorithm. Also, in the Discount algorithm the retention test and simplifications are sometimes performed twice on the same clause: the first time when the new clause is processed and then again when this clause is activated.

Comparison of Saturation Algorithms. Limited resource strategy is implemented only in VAMPIRE. Thus, VAMPIRE implements all three saturation algorithms. The theorem provers E [29] and Waldmeister [21] only implement the Discount algorithm. In VAMPIRE limited resource strategy gives the best results, closely followed by the

clauses	Otter and LRS	Discount
active	generating inferences with the given clause; simplifying inferences with new clauses	generating inferences with the given clause; simplifying inferences with new clauses; simplifying inferences with re-activated clauses
passive	simplifying inferences with new clauses	
new	simplifying inferences with active and passive clauses	simplifying inferences with active clauses

Fig. 6. Inferences in Saturation Algorithms

Discount algorithm. The Otter algorithm is generally weaker, but it still behaves better on some formulas.

To give the reader an idea on the difference between the three saturation algorithms, we show in Figure 6 the roles played by different clauses in the Otter and Discount algorithms.

5 Preprocessing

For many problems, preprocessing them in the right way is the key to solving them. VAMPIRE has a sophisticated preprocessor. An incomplete collection of preprocessing steps is listed below.

1. Select a *relevant* subset of formulas (optional).
2. Add *theory axioms* (optional).
3. *Rectify* the formula.
4. If the formula contains any occurrence of $\top$ or $\bot$, *simplify* the formula.
5. Remove *if-then-else* and *let-in* connectives.
6. *Flatten* the formula.
7. Apply *pure predicate elimination*.
8. Remove *unused predicate definitions* (optional).
9. Convert the formula into *equivalence negation normal form (ennf)*.
10. Use a *naming technique* to replace some subformulas by their names.
11. Convert the formula into *negation normal form* (optional).
12. *Skolemise* the formula.
13. Replace *equality axioms*.
14. Determine a *literal ordering* to be used.
15. Transform the formula into its *conjunctive normal form (cnf)*.
16. *Function definition elimination* (optional).
17. Apply *inequality splitting* (optional).
18. Remove *tautologies*.
19. Apply *pure literal elimination* (optional).
20. Remove *clausal definitions* (optional).

Optional steps are controlled by VAMPIRE's options. Since input problems can be very large (for example, several million formulas), VAMPIRE avoids using any algorithm that may cause slow preprocessing. Most algorithms used by the preprocessor run in linear or $O(n \cdot \log n)$ time.

6 Coloured Proofs, Interpolation, and Symbol Elimination

VAMPIRE can be used in program analysis, in particular for *loop invariant generation* and *building interpolants*. Both features have been implemented using our *symbol elimination method*, introduced in [18,19]. The main ingredient of symbol elimination is the use of *coloured proofs* and *local proofs*. Local proofs are also known as split proofs in [13] and they are implemented using coloured proofs. In this section we introduce interpolation, coloured proofs and symbol elimination and describe how they are implemented in VAMPIRE.

6.1 Interpolation

Let R and B be two closed formulas. Their *interpolant* is any formula I with the following properties:

1. $\vdash R \rightarrow I$;
2. $\vdash I \rightarrow B$;
3. Every symbol occurring in I also occurs both in R and B.

Note that the existence of an interpolant implies that $\vdash R \rightarrow B$, that is, B is a logical consequence of R. The first two properties mean that I is a formula "intermediate" in power between R and B. The third property means that I uses only (function and predicate) symbols occurring in both R and B. For example, suppose that p, q, r are propositional symbols. Then the formulas $p \wedge q$ and $q \vee r$ have an interpolant q.

Craig proved in [5] that every two formulas R and B such that $\vdash R \rightarrow B$ have an interpolant. In applications of interpolation in program verification, one is often interested in finding interpolants w.r.t. a theory T, where the provability relation $\vdash$ is replaced by $\vdash_T$, that is validity with respect to T. Interpolation in the presence of a theory T is discussed in some detail in [19].

6.2 Coloured Proofs and Symbol Elimination

We will reformulate the notion of an interpolant by *colouring* symbols and formulas. We assume to have three colour: *red*, *blue* and *grey*. Each symbol (function or predicate) is coloured in exactly one of these colours. Symbols that are coloured in red or blue are called *coloured symbols*. Thus, grey symbols are regarded as uncoloured. Similarly, a formula that contains at least one coloured symbols is called *coloured*; otherwise it is called *grey*. Note that coloured formulas can also contain grey symbols.

Let T be a theory and R and B be formulas such that

- each symbol in R is either red or grey;
- each symbol in B is either blue or grey;
- every formula in T is grey.

We call an *interpolant* of R and B any grey formula I such that $\vdash_T R \rightarrow I$ and $\vdash_T I \rightarrow B$.

It is easy to see that this definition generalises that of Craig [5]: use the empty theory, colour all symbols occurring in R but not in B in red, all symbols occurring in B but not in R blue, and symbols occurring in both R and B in grey.

When we deal with refutations rather than proofs and have an unsatisfiable set $\{R, B\}$, it is convenient to use a *reverse interpolant* of R and B, which is any grey formula I such that $\vdash_T R \rightarrow I$ and $\{I, B\}$ is unsatisfiable. In applications of interpolation in program verification, see e.g. the pioneering works [25,13], an interpolant is typically defined as a reverse interpolant. The use of interpolation in hardware and software verification requires deriving (reverse) interpolants from refutations.

A *local derivation* [13,19] is a derivation in which no inference contains both red and blue symbols. An inference with at least one coloured premise and a grey conclusion is called a *symbol-eliminating inference*. It turns out that one can extract interpolants from local proofs. For example, in [19] we gave an algorithm for extracting a reverse interpolant of R and B from a local refutation of $\{R, B\}$. The extracted reverse interpolant I is a boolean combination of conclusions of symbol-eliminating inferences, and is polynomial in the size of the refutation (represented as a dag). This algorithm is further extended in [11] to compute interpolants which are minimised w.r.t. various measures, such as the total number of symbols or quantifiers.

Coloured proofs can also be used for another interesting application of program verification. Suppose that we have a set Π of formulas in some language L and want to derive logical consequences of these formulas in a subset L_0 of this language. Then we declare the symbols occurring only in $L \setminus L_0$ coloured, say red, and the symbols of L_0 grey; this makes some of the formulas from Π coloured. We then ask VAMPIRE to eliminate red symbols from Π, that is, derive grey consequences of formulas in Π. All these grey consequences will be conclusions of symbol-eliminating inferences. This technique is called *symbol elimination* and was used in our experiments on automatic loop invariant generation [18,9]. It was the first ever method able to derive loop invariants with quantifier alternations. Our results [18,19] thus suggest that symbol elimination can be a unifying concept for several applications of theorem provers in program verification.

6.3 Interpolation and Symbol Elimination in VAMPIRE

To make VAMPIRE generate local proofs and compute an interpolant we should be able to assign colours to symbols and define which part of the input belongs to R, B, and the theory T, respectively. We will give an example showing how VAMPIRE implements coloured formulas, local proofs and generation of interpolants from local proofs.

Suppose that q, f, a, b are red symbols and c is a blue symbol, all other symbols are grey. Let R be the formula $q(f(a)) \wedge \neg q(f(b))$ and define B to be $(\exists v.v \neq c)$. Clearly, $\vdash R \rightarrow B$.

Specifying colours. The red and blue colours in Vampire are respectively denoted by `left` and `right`. To specify, for example, that the predicate symbol `q` of arity 1 is red and the constant `c` is blue, we use the following declarations:

```
vampire(symbol,predicate,q,1,left).
vampire(symbol,function,c,0,right).
```

Specifying R and B. Using the TPTP notation, formulas R and B are specified as:

```
vampire(left_formula).
  fof(R,axiom,q(f(a))&~q(f(b))).
vampire(end_formula).
```

```
vampire(right_formula).
  fof(L,conjecture,?[V].(V!=c)).
vampire(end_formula).
```

Local proofs and interpolants. To make VAMPIRE compute an interpolant I, the following option is set in the VAMPIRE input:

```
vampire(option,show_interpolant,on).
```

If we run VAMPIRE on this input problem, it will search for a local proof of $\vdash R \rightarrow B$. Such a local proof will quickly be found and the interpolant $\neg(\forall x, y)(x = y)$ will appear in the output.

Symbol elimination and invariants. To run VAMPIRE in the symbol elimination mode with the purpose of loop invariant generation, we declare the symbols to be eliminated coloured. For the use of symbol elimination, a single colour, for example `left`, is sufficient. We ask VAMPIRE to run symbol elimination on its coloured input by setting:

```
vampire(option,show_symbol_elimination,on).
```

When this option is on, VAMPIRE will output all conclusions of symbol-eliminating inferences.

7 Sorts and Theories

Standard superposition theorem provers are good in dealing with quantifiers but have essentially no support for theory reasoning. Combining first-order theorem proving and theory reasoning is very hard. For example, some simple fragments of predicate logic with linear arithmetic are already Π^1_1-complete [15]. One relatively simple way of combining first-order logic with theories is adding a first-order axiomatisation of the theory, for some example an (incomplete) axiomatisation of integers.

Adding incomplete axiomatisations is the approach used in [9,18] and the one we have followed in the development of VAMPIRE. We recently added integers, reals, and arrays as built-in data types in VAMPIRE, and extended VAMPIRE with such data types and theories. For example, one can use integer constants in the input instead of representing them using, for example, zero and the successor function. VAMPIRE implements several standard predicates and functions on integers and reals, including addition, subtraction, multiplication, successor, division, and standard inequality relations such as $\geq$. VAMPIRE also has an axiomatisations of the theory of arrays with the select and store operations.

7.1 Sorts

For using sorts in VAMPIRE, the user should first specify the sort of each symbol. Sort declarations need to be added to the input, by using the `tff` keyword ("typed first-order formula") of the TPTP syntax, as follows.

Defining sorts. For defining a new sort, say sort `own`, the following needs to be added to the VAMPIRE input:

```
tff(own_type,type,own: $tType).
```

Similarly to the `fof` declarations, the word `own_type` is chosen to denote the name of the declaration and is ignored by VAMPIRE when it searches for a proof. The keyword `type` in this declaration is, unfortunately, both ambiguous and redundant. The only important part of the declaration is `own: $tType`, which declares that `own` is a new sort (type).

Let us now consider an arbitrary constant a and a function f of arity 2. Specifying that a and the both the arguments and the values of f have sort `own` can be done as follows:

```
tff(a_has_type_own,type,a : own).
tff(f_has_type_own,type,f : own * own > own).
```

Pre-defined sorts. The user can also use the following pre-existing sorts of VAMPIRE:

- `$i`: sort of individuals. This is the default sort used in VAMPIRE: if a symbol is not declared, it has this sort;
- `$o`: sort of booleans;
- `$int`: sort of integers;
- `$rat`: sort of rationals;
- `$real`: sort of reals;
- `$array1`: sort of arrays of integers;
- `$array2`: sort of arrays of arrays of integers.

The last two are VAMPIRE-specific, all other sorts belong to the TPTP standard. For example, declaring that p is a unary predicate symbol over integers is written as:

```
tff(p_is_int_predicate,type,p : $int > $o).
```

7.2 Theory Reasoning

Theory reasoning in VAMPIRE is implemented by adding theory axioms to the input problem and using the superposition calculus to prove problems with both quantifiers and theories. In addition to the standard superposition calculus rules, VAMPIRE will evaluate expressions involving theory functions, whenever possible, during the proof search.

This implementation is not just a simple addition to VAMPIRE: we had to add simplification rules for theory terms and formulas, and special kinds of orderings [17]. Since VAMPIRE's users may not know much about combining theory reasoning and first-order logic, VAMPIRE adds the relevant theory axiomatisation automatically. For example, if the input problem contains the standard integer addition function symbol $+$, denoted by `$sum` in VAMPIRE, then VAMPIRE will automatically add an axiomatisation of integer linear arithmetic including axioms for additions.

The user can add her own axioms in addition to those added by Vampire. Moreover, the user can also choose to use her own axiomatisation *instead* of those added by VAMPIRE one by using the option `--theory axioms off`.

For some theories, namely for linear real and rational arithmetic, VAMPIRE also supports DPLL(T) style reasoning instead of using the superposition calculus, see Section 11.3. However, this feature is yet highly experimental.

A partial list of interpreted function and predicate symbols over integers/reals/rationals in VAMPIRE contains the following functions defined by the TPTP standard:

- `$sum`: addition $(x + y)$
- `$product`: multiplication $(x \cdot y)$
- `$difference`: difference $(x - y)$
- `$uminus`: unary minus $(-x)$
- `$to_rat`: conversion to rationals
- `$to_real`: conversion to reals
- `$less`: less than $(x < y)$
- `$lesseq`: less than or equal to $(x \leq y)$
- `$greater`: greater than $(x > y)$
- `$greatereq`: greater than or equal to $(x \geq y)$

Let us consider the formula $(x + y) \geq 0$, where x and y are integer variables. To make VAMPIRE try to prove this formula, one needs to use sort declarations in quantifiers and write down the formula as the typed first-order formula:

```
tff(example,conjecture, ? [X:$int,Y:$int]:
        $greatereq($sum(X,Y),0)).
```

When running VAMPIRE on the formula above, VAMPIRE will automatically load the theory axiomatisation of integers and interpret 0 as the corresponding integer constant.

The quantified variables in the above example are explicitly declared to have the sort `$int`. If the input contains undeclared variables, the TPTP standard requires that they have a predefined sort `$i`. Likewise, undeclared function and predicate symbols are considered symbols over the sort `$i`.

8 Satisfiability Checking Finding Finite Models

Since 2012, VAMPIRE has become competitive with the best solvers on satisfiable first-order problems. It can check satisfiability of a first-order formula or a set of clauses using three different approaches.

1. By saturation. If a complete strategy is used and VAMPIRE builds a saturated set, then the input set of formulas is satisfiable. Unfortunately, one cannot build a good representation of a model satisfying this set of clauses - the only witness for satisfiability in this case is the saturated set.
2. VAMPIRE implements the instance generation method of Ganzinger and Korovin [8]. It is triggered by using the option `--saturation_algorithm inst_gen`. If this method succeeds, the satisfiability witness will also be the saturated set.

Table 2. Strategies in VAMPIRE

category	strategies	total	best	worst	explanation
EPR	16	560	350	515	effectively propositional (decidable fragment)
UEQ	35	753	198	696	unit equality
HNE	22	536	223	469	CNF, Horn without equality
HEQ	23	436	376	131	CNF, Horn with equality
NNE	25	464	404	243	CNF, non-Horn without equality
NEQ	98	1861	1332	399	CNF, non-Horn with equality
PEQ	30	434	373	14	CNF, equality only, non-unit
FNE	34	1347	1210	585	first-order without equality
FEQ	193	4596	2866	511	first-order with equality

3. The method of building finite models using a translation to EPR formulas introduced in [3]. In this case, if satisfiability is detected, VAMPIRE will output a representation of the found finite model.

Finally, one can use `--mode casc_sat` to treat the input problem using a cocktail of satisfiability-checking strategies.

9 VAMPIRE Options and the CASC Mode

VAMPIRE has many *options* (parameter-value pairs) whose values that can be changed by the user (in the command line). By changing values of the options, the user can control the input, preprocessing, output of proofs, statistics and, most importantly, proof search of VAMPIRE. A collection of options is called a *strategy*. Changing values of some parameters may have a drastic influence on the proof search space. It is not unusual that one strategy results in a refutation found immediately, while for another strategies VAMPIRE cannot find refutation in hours.

When one runs VAMPIRE on a problem, the default strategy of VAMPIRE is used. This strategy was carefully selected to solve a reasonably large number of problems based on the statistics collected by running VAMPIRE on problems from the TPTP library. However, it is important to understand that there is no single best strategy, so for solving hard problems using a single strategy is not a good idea. Table 2 illustrates the power of strategies, based on the results of running VAMPIRE using various strategies on several problem categories. The acronyms for categories use the TPTP convention, for example FEQ means "first-order problems with equality", their explanation is given in the rightmost column. The table respectively shows the total number of strategies used in experiments, the total number of problems solved by all strategies, and the numbers of problems solved by the best and the worst strategy. For example, the total number of TPTP FEQ problems in this category that VAMPIRE can solve is 4596, while the best strategy for this category can solve only 2866 problems, that is, only about 62% of all problems. The worst strategy solves only 511 FEQ problems, but this strategy is highly incomplete.

There is an infinite number of possible strategies, since some parameters have integer values. Even if we fix a small finite number of values for such parameters, the total

```
Hi Geoff, go and have some cold beer while I am trying to solve
                                          this very hard problem!
% remaining time: 2999 next slice time: 7
dis+2_64_bs=off:cond=fast:drc=off:fsr=off:lcm=reverse:nwc=4:...
Time limit reached!
------------------------------
Termination reason: Time limit
...
------------------------------
% remaining time: 2991 next slice time: 7
dis+10_14_bs=off:cond=fast:drc=off:gs=on:nwc=2.5:nicw=on:sd=...
Refutation not found, incomplete strategy
------------------------------
Termination reason: Refutation not found, incomplete strategy
...
------------------------------
% remaining time: 2991 next slice time: 18
dis+1011_24_cond=fast:drc=off:nwc=10:nicw=on:ptb=off:ssec=of...
Refutation found. Thanks to Tanya!
...
% Success in time 0.816 s
```

Fig. 7. Running VAMPIRE in the CASC mode on SET014-3

number of possible strategies will be huge. Fortunately, VAMPIRE users do not have to understand all the parameters. One can use a special VAMPIRE mode, called the CASC mode and used as `--mode casc`, which mimics the strategies used at the last CASC competition [32]. In this mode, VAMPIRE will treat the input problem with a cocktail of strategies running them sequentially with various time limits.

The CASC mode of VAMPIRE is illustrated in Figure 7 and shows part of the output produced by VAMPIRE in the CASC mode on the TPTP problem `SET014-3`. This problem is very hard: according to the TPTP problems and solutions document only VAMPIRE can solve it. One can see that VAMPIRE ran three different strategies on this problems. Each of the strategies is given as a string showing in a succinct way the options used. For example, the string `dis+2_64_bs=off:cond=fast` means that VAMPIRE was run using the Discount saturation algorithm with literal selection 2 and age-weight ratio 64. Backward subsumption was turned off and a fast condensing algorithm was used. In the figure, we truncated these strings since some of them are very long and removed statistics (output after running each strategy) and proof (output at the end).

The time in the CASC mode output is measured in deciseconds. One can see that VAMPIRE used the first strategy with the time limit of 0.7 seconds, then the second one with the time limit of 0.7 seconds, followed by the third strategy with the time limit of 1.8 seconds. All together it took VAMPIRE 0.816 seconds to find a refutation.

There is also a similar option for checking satisfiability: `--mode casc_sat`. The use of the CASC mode options, together with a time limit, is highly recommended.

```
Precondition:
{(∀ X) (p(X) => X
≥ 0)}
{(∀ X) (f(X) > 0)}
{p(a)}

Program:
if (q(a))
{ a := a+1 }
else
{ a := a + f(a)}

Postcondition:
{a > 0}
```

```
% sort declarations
tff(1,type,p : $int > $o).
tff(2,type,f : $int > $int).
tff(3,type,q : $int > $o).
tff(4,type,a : $int).

% precondition
tff(5,hypothesis,
  ! [X:$int] : (p(X) => $greatereq(X,0))).
tff(6,hypothesis,
  ! [X:$int] : ($greatereq(f(X),0))).
tff(7,hypothesis,p(a)).

% transition relation
tff(8,hypothesis,
  a1 = $ite_t(q(a),
              $let_tt(a,$sum(a,1),a),
              $let_tt(a,$sum(a,f(a)),a))).

% postcondition
tff(9,conjecture,$greater(a1,0)).
```

Fig. 8. A partial correctness statement and its VAMPIRE input representation

The total number of parameters in the current version of VAMPIRE is 158, so we cannot describe all of them here. These options are not only related to the proof search. There are options for preprocessing, input and output syntax, output of statistics, various proof search limit, interpolation, SAT solving, program analysis, splitting, literal and clause selection, proof output, syntax of new names, bound propagation, use of theories, and some other options.

10 Advanced TPTP Syntax

VAMPIRE supports let-in and if-then-else constructs, which have recently become a TPTP standard. VAMPIRE was the first ever first-order theorem prover to implement these constructs. Having them makes VAMPIRE better suited for applications in program verification. For example, given a simple program with assignments, composition and if-then-else, one can easily express the transition relation of this programs.

Consider the example shown in Figure 8. The left column displays a simple program together with its pre- and postconditions. Let `a1` denote the next state value of `a`. Using if-then-else and let-in expressions, we can express this next-state value by a simple transformation of the text of the program as follows:

```
a1 = if q(a) then (let a=a+1 in a)
             else (let a=a+f(a) in a)
```

We can then express partial correctness of the program of Figure 8 as a TPTP problem, as follows:

- write down the precondition as a hypothesis in the TPTP syntax;
- write down the next state value of `a` as as a hypothesis in the extended TPTP syntax, by using the if-then-else (`$ite_t`) and let-in (`$let_tt`) constructs;
- write down the postcondition as the conjecture.

The right column of Figure 8 shows this TPTP encoding.

11 Cookies

One can use VAMPIRE not only as a theorem prover, but in several other ways. Some of them are described in this section.

11.1 Consequence Elimination

Given a large set S of formulas, it is very likely that some formulas are consequences of other formulas in the set. To simplify reasoning about and with S, it is desirable to simplify S by removing formulas from S that are implied by other formulas. To address this problem, a new mode, called *the consequence-elimination mode* is now added to VAMPIRE [10]. In this mode, VAMPIRE takes a set S of clauses as an input and tries to find its proper subset S_0 equivalent to S. In the process of computing S_0, VAMPIRE is run with a small time limit. Naturally, one is interested in having S_0 as small as possible. To use VAMPIRE for consequence elimination, one should run:

```
vampire --mode consequence elimination set_S.tptp
```

where `set_S.tptp` contains the set S.

We illustrate consequence elimination on the following set of formulas:

```
fof(ax1, axiom, a => b).
fof(ax2, axiom, b => c).
fof(ax3, axiom, c => a).
fof(c1, claim, a | d).
fof(c2, claim, b | d).
fof(c3, claim, c | d).
```

The word `claim` is a new VAMPIRE keyword for input formulas introduced for the purpose of consequence elimination. VAMPIRE is asked to eliminate, one by one, those claims that are implied by other claims and axioms. The set of non-eliminated claims will then be equivalent to the set of original claims (modulo axioms).

In this example, the axioms imply that formulas `a`, `b` and `c` are pairwise equivalent, hence all claims are equivalent modulo the axioms. VAMPIRE detects that the formula named `c2` is implied by `c1` (using axiom `ax2`), and then that `c1` is implied by `c3` (using axiom `ax3`). Formulas `c1` and `c2` are thus removed from the claims, resulting in the simplified set of claims containing only `c3`, that is, `c | d`.

```
while (a ≤ m) do
  if A[a] ≥ 0
    then B[b] := A[a];b := b + 1;
    else C[c] := A[a];c := c + 1;
  a := a + 1;
end do
```

Fig. 9. Array partition

Consequence elimination turns out to be very useful for pruning a set of automatically generated loop invariants. For example, symbol elimination can generate invariants implied by other generated invariants. For this reason, in the program analysis mode of VAMPIRE, described in the next section, symbol elimination is augmented by consequence elimination to derive a minimised set of loop invariants.

11.2 Program Analysis

Starting with 2011, VAMPIRE can be used to parse and analyse software programs written in a subset of C, and generate properties of loops in these programs using the symbol elimination method introduced in [10]. The subset of C consists of scalar integer variables, array variables, arithmetical expressions, assignments, conditionals and loops. Nested loops are yet unsupported. Running VAMPIRE in the program analysis mode can be done by using:

```
vampire --mode program_analysis problem.c
```

where `problem.c` is the C program to be analysed.

We illustrate the program analysis part of VAMPIRE on Figure 9. This program respectively fills the arrays B and C with the non-negative and negative values of the source array A. Figure 10 shows a (slightly modified) partial output of VAMPIRE's program analysis on this program. VAMPIRE first extracts all loops from the input program, ignores nested loops and analyses every non-nested loop separately. In our example, only one loop is found (also shown in Figure 10). The following steps are then performed for analysing each of the loops:

1. *Find all loop variables* and classify them into variables updated by the loop and constant variables. In Figure 10, the program analyser of VAMPIRE detects that variables B, C, a, b, c are updated in the loop, and variables A, m are constants.
2. *Find counters*, that is, updated scalar variables that are only incremented or decremented by constant values. Note that expressions used as array indexes in loops are typically counters. In Figure 10, the variables a, b, c are classified as counters.
3. *Infer properties of counters*. VAMPIRE adds a new constant `'$counter'` denoting the "current value" of the loop counter, and program variables updated in the loop become functions of the loop counter.

   For example, the variable a becomes the unary function $a(X)$, where $a(X)$ denotes the value of a at the loop iteration $X \leq$ `'$counter'`. The value $a(0)$

```
Loops found: 1
Analyzing loop...
-------------------
while (a < m)
{
 if (A[a] >= 0)
 {
    B[b] = A[a];
    b = b + 1;
 }
 else
 {
    C[c] = A[a];
    c = c + 1;
 }
  a = a + 1;
}
-------------------
Variable: B updated
Variable: a updated
Variable: b updated
Variable: m constant
Variable: A constant
Variable: C updated
Variable: c updated
Counter: a
Counter: b
Counter: c
```

```
Collected first-order loop properties
-------------------

27. iter(X0) <=> ($lesseq(0,X0) & $less(X0,'$counter'))

17. a(0) = a0

10. updbb(X0,X2,X3) => bb(X2) = X3

9. updbb(X0,X2,X3) <=>
(let b := b(X0) in (let c := c(X0) in (let a := a(X0) in
(let cc(X1) := cc(X0,X1) in (let bb(X1) := bb(X0,X1) in
($greatereq(aa(a),0) & (aa(a) = X3 & iter(X0) & b = X2)))))))

8. updbb(X0,X2) <=>
(let b := b(X0) in (let c := c(X0) in (let a := a(X0) in
(let cc(X1) := cc(X0,X1) in  (let bb(X1) := bb(X0,X1) in
($greatereq(aa(a),0) & (iter(X0) & b = X2)))))))

4.
($greater(X1,X0) & $greater(c(X1),X3) & $greater(X3,c(X0)))
=> ? [X2] : (c(X2) = X3 & $greater(X2,X0) & $greater(X1,X2))

3. $greatereq(X1,X0) => $greatereq(c(X1),c(X0))

1. a(X0) = $sum(a0,X0)
```

Fig. 10. Partial output of VAMPIRE's program analysis

thus denotes the initial value of the program variable a. Since we are interested in loop properties connecting the initial and the current value of the loop variable, we introduce a constant $a0$ denoting the initial value of a (see formula `17`). The predicate $iter(X)$ in formula `27` defines that X is a loop iteration. The properties inferred at this step include formulas `1`, `3` and `4`. For example, formula 3 states that c is a monotonically increasing variable.

4. *Generate update predicates* of updated array variables, for example formula `10`. The update predicate `updbb(X0,X1)` expresses that the array B was updated at loop iteration $X0$ at position $X1$ by value $X2$. The generated properties are in the TPTP syntax, a reason why the capitalised array variables are renamed by VAMPIRE's program analyser. For example, the array B is denoted by bb.
5. Derive the formulas corresponding to *the transition relation of the loop* by using let-in and if-then-else constructs. These properties include, for example, formulas `8` and `9`.
6. Generate a symbol elimination task for VAMPIRE by colouring symbols that should not be used in loop invariant. Such symbols are, for example, `'$counter'`, `iter`, and `updbb`.
7. Run VAMPIRE in the consequence elimination mode and output a minimised set of loop invariants. As a result of running VAMPIRE's program analyser on Figure 9, one of the invariants derived by VAMPIRE is the following first-order formula:

```
tff(inv,claim, ![X:$int]:
($greatereq(X,0) & $greater(b,X) => $greatereq(bb(X),0) &
(? [Y:$int]: $greatereq(Y,0) & $greater(a,Y) & aa(Y)=bb(X) )))
```

11.3 Bound Propagation

We have recently extended VAMPIRE with a decision procedure for quantifier-free linear real and rational arithmetic. Our implementation is based on the bound propagation algorithm of [16], BPA in the sequel. When VAMPIRE is run in the BPA mode on a system of linear inequalities, VAMPIRE tries to solve these inequalities by applying the BPA algorithm instead of the superposition calculus. If the system is found to be satisfiable, VAMPIRE outputs a solution. To run VAMPIRE in the BPA mode, one should use:

```
vampire --mode bpa problem.smt
```

where `problem.smt` is a set of linear real and rational inequalities represented in the SMTLIB format [2]. We also implemented various options for changing the input syntax and the representation of real and rational numbers in VAMPIRE; we refer to [36] for details. Our BPA implementation in VAMPIRE is the first step towards combining superposition theorem proving with SMT style reasoning.

11.4 Clausifier

VAMPIRE has a very fast TPTP parser and clausifier. One can use VAMPIRE simply to *clausify formulas*, that is, convert them to clausal normal form. To this end, one should use `--mode clausify`. Note that for clausification one can use all VAMPIRE's options for preprocessing. VAMPIRE will output the result in the TPTP syntax, so that the result can be processed by any other prover understanding the TPTP language.

11.5 Grounding

One can use VAMPIRE to *convert an EPR problem to a SAT problem*. This problem will be output in the DIMACS format. To this end one uses the option `--mode grounding`. Note that VAMPIRE will not try to optimise the resulting set of propositional clauses in any way, so this feature is highly experimental.

12 Conclusions

We gave a brief introduction to first-theorem proving and discussed the use and implementation of VAMPIRE. VAMPIRE is fully automatic and can be used in various applications of automated reasoning, including theorem proving, first-order satisfiability checking, finite model finding, and reasoning with both theories and quantifiers. We also described our recent developments of new and unconventional applications of theorem proving in program verification, including interpolation, loop invariant generation, and program analysis.

References

1. Bachmair, L., Ganzinger, H.: Resolution theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 2, vol. I, pp. 19–99. Elsevier Science (2001)
2. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (2010), http://www.SMT-LIB.org
3. Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing Finite Models by Reduction to Function-Free Clause Logic. J. of Applied Logic 7(1), 58–74 (2009)
4. Buchberger, B.: An Algorithm for Finding the Basis Elements of the Residue Class Ring of a Zero Dimensional Polynomial Ideal. J. of Symbolic Computation 41(3-4), 475–511 (2006); Phd thesis 1965, University of Innsbruck, Austria
5. Craig, W.: Three uses of the Herbrand-Gentzen Theorem in Relating Model Theory and Proof Theory. J. of Symbolic Logic 22(3), 269–285 (1957)
6. Denzinger, J., Kronenburg, M., Schulz, S.: DISCOUNT — A Distributed and Learning Equational Prover. J. of Automated Reasoning 18(2), 189–198 (1997)
7. Dershowitz, N., Plaisted, D.A.: Rewriting. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 9, vol. I, pp. 535–610. Elsevier Science (2001)
8. Ganzinger, H., Korovin, K.: New Directions in Instantiation-Based Theorem Proving. In: Proc. of LICS, pp. 55–64 (2003)
9. Hoder, K., Kovács, L., Voronkov, A.: Case Studies on Invariant Generation Using a Saturation Theorem Prover. In: Batyrshin, I., Sidorov, G. (eds.) MICAI 2011, Part I. LNCS, vol. 7094, pp. 1–15. Springer, Heidelberg (2011)
10. Hoder, K., Kovács, L., Voronkov, A.: Invariant Generation in Vampire. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 60–64. Springer, Heidelberg (2011)
11. Hoder, K., Kovács, L., Voronkov, A.: Playing in the Grey Area of Proofs. In: Proc. of POPL, pp. 259–272 (2012)
12. Hoder, K., Voronkov, A.: Comparing Unification Algorithms in First-Order Theorem Proving. In: Mertsching, B., Hund, M., Aziz, Z. (eds.) KI 2009. LNCS, vol. 5803, pp. 435–443. Springer, Heidelberg (2009)
13. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
14. Knuth, D., Bendix, P.: Simple Word Problems in Universal Algebras. In: Leech, J. (ed.) Computational Problems in Abstract Algebra, pp. 263–297. Pergamon Press (1970)
15. Korovin, K., Voronkov, A.: Integrating Linear Arithmetic into Superposition Calculus. In: Duparc, J., Henzinger, T.A. (eds.) CSL 2007. LNCS, vol. 4646, pp. 223–237. Springer, Heidelberg (2007)
16. Korovin, K., Voronkov, A.: Solving Systems of Linear Inequalities by Bound Propagation. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 369–383. Springer, Heidelberg (2011)
17. Kovács, L., Moser, G., Voronkov, A.: On Transfinite Knuth-Bendix Orders. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 384–399. Springer, Heidelberg (2011)
18. Kovács, L., Voronkov, A.: Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
19. Kovács, L., Voronkov, A.: Interpolation and Symbol Elimination. In: Schmidt, R.A. (ed.) CADE-22. LNCS, vol. 5663, pp. 199–213. Springer, Heidelberg (2009)
20. Kovács, L., Voronkov, A.: Vampire Web Page (2013), http://vprover.org

21. Löchner, B., Hillenbrand, T.: A Phytography of WALDMEISTER. AI Commun. 15(2-3), 127–133 (2002)
22. Lusk, E.L.: Controlling Redundancy in Large Search Spaces: Argonne-Style Theorem Proving Through the Years. In: Proc. of LPAR, pp. 96–106 (1992)
23. Martelli, A., Montanari, U.: An Efficient Unification Algorithm. TOPLAS 4(2), 258–282 (1982)
24. McCune, W.W.: OTTER 3.0 Reference Manual and Guide. Technical Report ANL-94/6, Argonne National Laboratory (January 1994)
25. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
26. Nieuwenhuis, R., Rubio, A.: Paramodulation-based theorem proving. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 7, vol. I, pp. 371–443. Elsevier Science (2001)
27. Riazanov, A., Voronkov, A.: Limited resource strategy in resolution theorem proving. Journal of Symbolic Computations 36(1-2), 101–115 (2003)
28. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. J. ACM 12(1), 23–41 (1965)
29. Schulz, S.: E — a Brainiac Theorem Prover. AI Commun. 15(2-3), 111–126 (2002)
30. Sekar, R., Ramakrishnan, I.V., Voronkov, A.: Term indexing. In: Robinson, A., Voronkov, A. (eds.) Handbook of Automated Reasoning, ch. 26, vol. II, pp. 1853–1964. Elsevier Science (2001)
31. Sutcliffe, G.: The TPTP Problem Library and Associated Infrastructure. J. Autom. Reasoning 43(4), 337–362 (2009)
32. Sutcliffe, G.: The CADE-23 Automated Theorem Proving System Competition - CASC-23. AI Commun. 25(1), 49–63 (2012)
33. Sutcliffe, G.: The TPTP Problem Library (2013), `http://www.cs.miami.edu/~tptp/`
34. Sutcliffe, G.: The CADE ATP System Competition (2013), `http://www.cs.miami.edu/~tptp/CASC/`
35. Weidenbach, C., Gaede, B., Rock, G.: SPASS & FLOTTER. Version 0.42. In: McRobbie, M.A., Slaney, J.K. (eds.) CADE 1996. LNCS, vol. 1104, pp. 141–145. Springer, Heidelberg (1996)
36. `http://www.complang.tuwien.ac.at/ioan/boundPropagation`. Vampire with Bound Propagation

Software Model Checking for People Who Love Automata

Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski

University of Freiburg, Germany

Abstract. In this expository paper, we use automata for software model checking in a new way. The starting point is to fix the alphabet: the set of statements of the given program. We show how automata over the alphabet of statements can help to decompose the main problem in software model checking, which is to find the right abstraction of a program for a given correctness property.

1 Introduction

Automata provide the algorithmic basis in many applications. In particular, we can use automata-based algorithms to implement a data structure for operations over *sets of sequences*. An automaton defines a set of sequences over some alphabet. Or, in the terminology of formal language theory: an automaton *recognizes a language* (the elements in the sequence are *letters*, a sequence of letters is a *word*, a set of words is a *language*).

Formally, an automaton is a finite graph; its edges are labeled by letters of the alphabet; an initial node and a set of final nodes are distinguished among its nodes. The labeling of a path is a word. The automaton defines the set of all words that label a path from the initial node to one of the final nodes.

In this expository paper, we use automata for software model checking in a new way. The starting point is to fix the alphabet: the set of statements of the given program. The idea that a statement is a letter may take some time to get used to. As a letter, a statement is deprived of its meaning; the only purpose of a letter is to be used in a word. We are not used to freely compose statements to words, regardless of whether the word makes any sense as a sequence of statements or not.

In software model checking, a (if not *the*) central problem is to automatically find the right abstraction of a program for a given correctness property. In the remainder of this section, we will use three examples to illustrate how automata over the alphabet of statements can help to automatically decompose this problem. Then, in Section 2, we will fix the formal setting that allows us to relate the correctness of programs with automata over the alphabet of statements. In Section 3, we will define the notion of *Floyd-Hoare automata* for a given correctness property, and we will present different ways to construct such automata. We will see that, for a given program, the construction of Floyd-Hoare automata can be used to automatically decompose the task of finding the right abstraction of the program for the given correctness property.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 36–52, 2013.

```
ℓ0: assume p != 0;

ℓ1: while(n >= 0)
    {
ℓ2:     assert p != 0;

        if(n == 0)
        {
ℓ3:         p := 0;
        }
ℓ4:     n--;
    }
```

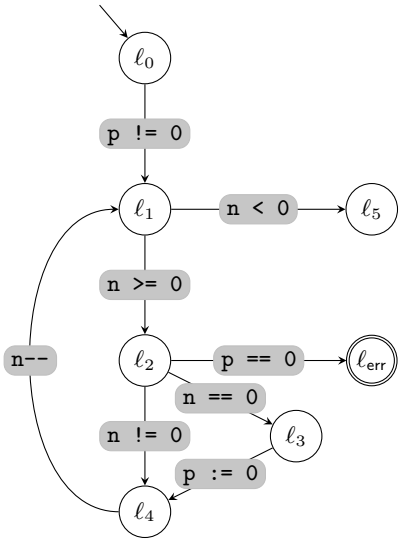


Fig. 1. Example program $\mathcal{P}_{\text{ex1}}$

Example 1: Automata from Infeasibility Proofs

The program $\mathcal{P}_{\text{ex1}}$ in Figure 1 is the adaptation of an example in [15] to our setting. In our setting we use `assert` statements to define the correctness of the program executions. In the example of $\mathcal{P}_{\text{ex1}}$, an *incorrect* execution would start with a non-zero value for the variable `p` and, at some point, enter the body of the while loop when the value of `p` is 0 (and the execution of the `assert` statement *fails*).

We can argue the correctness of $\mathcal{P}_{\text{ex1}}$ rather directly if we split the executions into two cases, namely according to whether the `then` branch of the conditional gets executed at least once during the execution or it does not. If not, then the value of `p` is never changed and remains non-zero (and the assert statement cannot fail). If the `then` branch of the conditional is executed, then the value of `n` is 0, the statement `n--` decrements the value of `n` from 0 to -1, and the while loop will exit directly, without executing the `assert` statement.

We can infer a case split like the one above automatically. The key is to use automata. For one thing, we can use automata as an expressive means to characterize different cases of execution paths. For another, instead of first fixing the case split and then constructing the corrresponding correctness arguments, we can construct an automaton for a given correctness argument so that the automaton characterizes the case of exactly the executions for which the correctness argument applies. We will next illustrate this in the example of $\mathcal{P}_{\text{ex1}}$.

We will describe an execution of $\mathcal{P}_{\text{ex1}}$ through the sequence of statements on the corresponding path in the *control flow graph* of $\mathcal{P}_{\text{ex1}}$; see Figure 1. The shortest path from ℓ_0 to ℓ_{err} goes via ℓ_1 and ℓ_2. The sequence of statements on this path is *infeasible* (it does not have a possible execution) because it is not

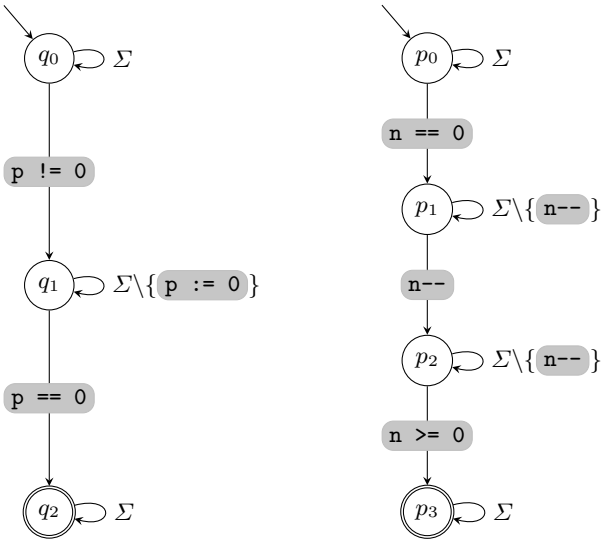


Fig. 2. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ which are a proof of correctness for $\mathcal{P}_{\text{ex1}}$ (an edge labelled with Σ means a transition reading any letter, an edge labeled with $\Sigma\backslash\{$ `p := 0` $\}$) means a transition reading any letter except for `p := 0`, etc.)

possible to execute the `assume` statements `p!=0` and `p==0` without an update of `p` in between.

We construct the automaton $\mathcal{A}_1$ in Figure 2 which recognizes the set of all sequences of statements that contain `p!=0` and `p==0` without an update of `p` in between (and with any statements before or after). I.e., $\mathcal{A}_1$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of $p \neq 0$ and $p = 0$).

A sequence of statements is *not* accepted by $\mathcal{A}_1$ if it contains `p!=0` and `p==0` *with* an update of `p` in between. The shortest path from ℓ_0 to ℓ_{err} with such a sequence of statements goes from ℓ_2 to ℓ_{err} after it has gone from ℓ_2 to ℓ_3 once before. The sequence of statements on this path is infeasible for a new reason: it is not possible to execute the `assume` statement `n==0`, the update statement `n--`, and the `assume` statement `n>=0` unless there is an (other) update of `n` between `n==0` and `n--` or between `n--` and `n>=0`.

We construct the automaton $\mathcal{A}_2$ depicted in Figure 2 which recognizes the set of all sequences of statements that contain the statements `n==0`, `n--`, and `n>=0` without an update of `n` in between (and with any statements before or after). I.e., $\mathcal{A}_2$ recognizes the set of sequences of statements that are infeasible for the same reason as above (i.e., the inconsistency of the three conjuncts $n = 0$, $n' = n - 1$, and $n' \geq 0$).

To summarize, we have twice taken a path from ℓ_0 to ℓ_{err}, analyzed the reason of its infeasibility, and constructed an automaton which each recognizes the set

of sequences of statements that are infeasible for the specific reason. The two automata thus characterize a case of executions in the sense discussed above.

Can one automatically check that every possible execution of $\mathcal{P}_{\text{ex1}}$ falls into one of the two cases? – The corresponding decision problem is undecidable. We can, however, check a condition which is stronger, namely that all sequences of statements on paths from ℓ_0 to ℓ_{err} in the control flow graph of $\mathcal{P}_{\text{ex1}}$ fall into one of the two cases (the condition is stronger because not every such path corresponds to a possible execution). The set of such sequences is the language recognized by an automaton which we also call $\mathcal{P}_{\text{ex1}}$ (recall that an automaton accepts a word exactly if the word labels a path from the initial state to a final state). Thus, the check amounts to checking the inclusion between automata, namely

$$\mathcal{P}_{\text{ex1}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2.$$

To rephrase our summary in the terminology of automata, we have twice taken a word accepted by the automaton $\mathcal{P}_{\text{ex1}}$, we have analyzed the reason of the infeasibility of the word (i.e., the corresponding sequence of statements), and we have constructed an automaton which recognizes the set of all words for which the same reason applies.

The view of a program as an automaton over the alphabet of statements may take some time to get used to because the view ignores the operational meaning of the program.

Example 2: Automata from Sets of Hoare Triples

It is "easy" to justify the construction of the automata $\mathcal{A}_1$ and $\mathcal{A}_2$ in Example 1: the infeasibility of a sequence of statements (such as the sequence `p!=0 p==0`) is preserved if one adds statements that do not modify any of the variables of the statements in the sequence (here, the variable `p`).

The example of the program $\mathcal{P}_{\text{ex2}}$ in Figure 3 shows that sometimes a more involved justification is required. The sequence of the two statements `x:=0` and `x==-1` (which labels a path from ℓ_0 to ℓ_{err}) is infeasible. However, the statement `x++` does modify the variable that appears in the two statements. So how can we account for the paths that loop in ℓ_2 taking the edge labeled `x++` one or more times? We need to construct an automaton that covers the case of those paths, but we cannot base the construction solely on infeasibility (as we did in Example 1).

We must base the construction of the automaton on a more powerful form of correctness argument: Hoare triples. The four Hoare triples below are sufficient to prove the infeasibility of all those paths. They express that the assertion $x \geq 0$ holds after the update `x:=0`, that it is *invariant* under the updates `y:=0` and `x++`, and that is blocks the execution of the assume statement `x==-1`.

$$\begin{array}{llll}
\{\ \mathit{true}\ \} & \texttt{x:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{y:=0} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x++} & \{x \geq 0\} \\
\{x \geq 0\} & \texttt{x==-1} & \{\ \mathit{false}\ \}
\end{array}$$

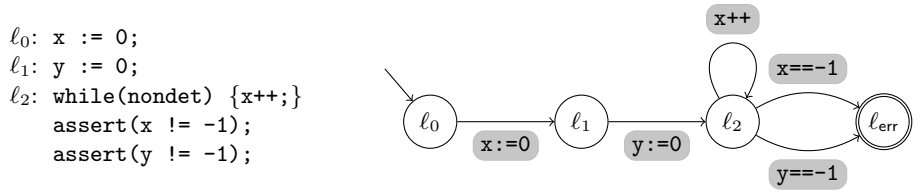


Fig. 3. Example program $\mathcal{P}_{\text{ex2}}$

The automaton $\mathcal{A}_1$ in Figure 4 has four transitions, one for each Hoare triple. It has three states, one for each assertion: the initial state q_0 for *true*, the state q_1 for $x \geq 0$, the (only) final state q_2 for *false*. The construction of such a *Floyd-Hoare automaton* generalizes to any set of Hoare triples. The resulting automaton can have arbitrary loops. In contrast, an automaton constructed as in the preceding example can only have self-loops.

Where does the set of Hoare triples come from? In this example, it may come from a static analysis [7] applied to the program fragment that corresponds to one path from ℓ_0 to ℓ_{err}; such a static analysis may assign an abstract value corresponding to $x \geq 0$ to the location ℓ_2 and determine that ℓ_{err} is not reachable.

In our implementation [11], the set of Hoare triples comes from an interpolating SMT solver [5] which generates the assertion $x \geq 0$ from the infeasibility proof.

The four Hoare triples below are sufficient to prove the infeasibility of all paths that reach the error location via the edge labeled with `y==-1`.

$$\begin{array}{l} \{\ \mathit{true}\ \}\ \texttt{x:=0}\ \{\ \mathit{true}\ \} \\ \{\ \mathit{true}\ \}\ \texttt{y:=0}\ \{y = 0\} \\ \{y = 0\}\ \ \texttt{x++}\ \ \{y = 0\} \\ \{y = 0\}\ \texttt{y==-1}\ \{\ \mathit{false}\ \} \end{array}$$

We use them in the same way as above in order to construct the automaton $\mathcal{A}_2$ in Figure 4. The two automata are sufficient to prove the correctness of the program; i.e., $\mathcal{P}_{\text{ex2}} \subseteq \mathcal{A}_1 \cup \mathcal{A}_2$.

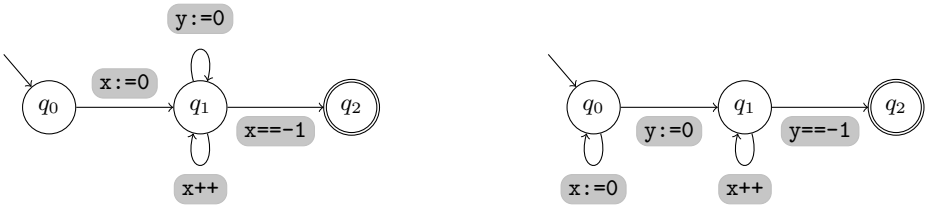


Fig. 4. Automata $\mathcal{A}_1$ and $\mathcal{A}_2$ for $\mathcal{P}_{\text{ex2}}$

The Hoare triple $\{y = 0\}$ `x++` $\{y = 0\}$ holds trivially. This is related to the fact that the statement `x++` does not modify the variable of the statements in the infeasible sequence `y:=0 y==-1`. I.e., we could have based the construction of the automaton $\mathcal{A}_2$ in Figure 4 on an infeasibility proof as for the automata in Example 1.

Example 3: Automata for Trace Partitioning

The $\mathcal{P}_{\text{ex3}}$ in Figure 5 is a classical example used to motivate *trace partitioning* for static analysis (see, e.g., [18]). As shown in Figure 5, an interval analysis applied to the program will derive that the value of the variable `x` at location ℓ_3 lies in the interval $[-1, 1]$. This is not sufficient to prove that the error location is not reachable. One remedy is to *partition* the executions into two cases according to whether the execution takes the `then` or the `else` branch of the conditional. The static analysis applied to each of the two cases separately will derive that the value of the variable `x` at location ℓ_3 lies in the intervall $[-1, -1]$ (in the `else` case) or in the interval $[1, 1]$ (in the `then` case). In either case, the static analysis derives the unreachability of the error location.

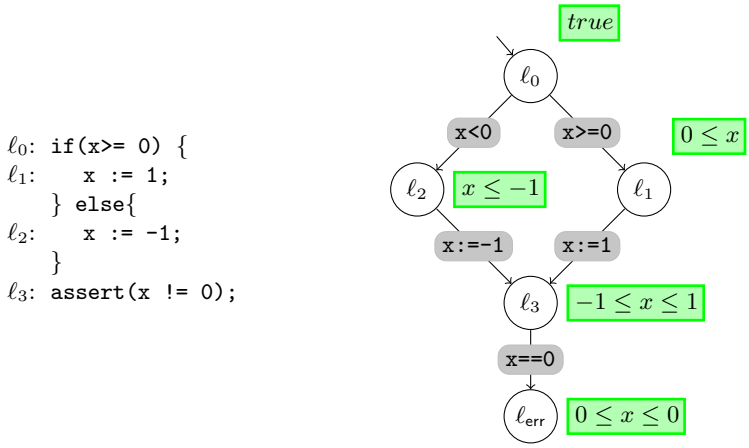
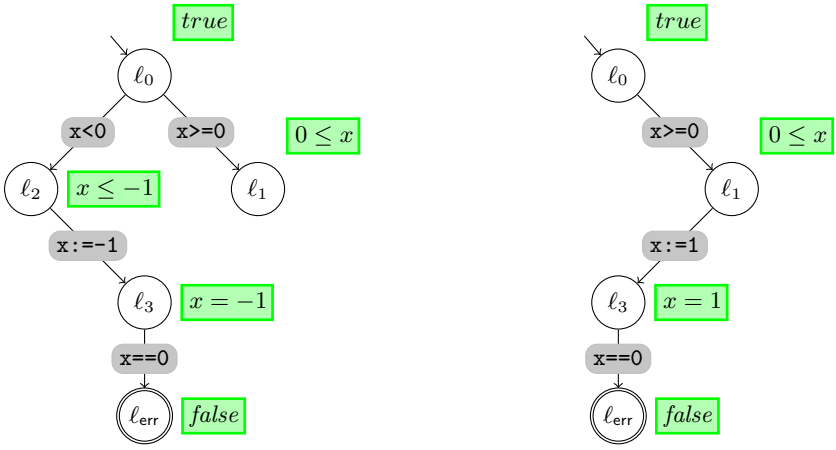


Fig. 5. Example program $\mathcal{P}_{\text{ex3}}$ (the labeling of program locations with assertions translates the result of an interval analysis; the labeling of the error location is not the assertion *false* which means that the interval analysis does not prove that the error location is unreachable; for each edge between two nodes, the two assertions and the statement form a Hoare triple)

We will use the example to illustrate how automata can be used to infer this kind of partitioning automatically for a given verification task.

Consider the partial annotation shown in Figure 6a. As in Figure 5, each edge corresponds to a Hoare triple, but there is no edge from ℓ_1 to ℓ_3.



(a) $\mathcal{P}^1_{\text{ex3}}$ **(b)** $\mathcal{P}^2_{\text{ex3}}$

Fig. 6. Programs $\mathcal{P}^1_{\text{ex3}}$ and $\mathcal{P}^2_{\text{ex3}}$ obtained by automata-based trace partitioning (the labeling of program locations with assertions translates again the result of the interval analysis; the programs are defined by $\mathcal{P}^1_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \cap \mathcal{A}_1$ and $\mathcal{P}^2_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \setminus \mathcal{A}_1$ where the automaton $\mathcal{A}_1$ is constructed from an intermediate result of the interval analysis applied to $\mathcal{P}_{\text{ex3}}$; the intermediate result provides the assertions $x = -1$ for ℓ_3 and *false* for ℓ_{err} but does not provide a Hoare triple for the edge from ℓ_1 to ℓ_3; the executions of $\mathcal{P}^2_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have not yet been proven correct by the intermediate result of an interval analysis applied to $\mathcal{P}_{\text{ex3}}$)

The partial annotation is established by taking the intermediate results of the static analysis. This includes in particular the Hoare triples $\{x \leq -1\}$ `x:=-1` $\{x = -1\}$ and $\{x \leq -1\}$ `x==0` $\{\mathit{false}\}$. We construct the automaton $\mathcal{A}_1$ from the set of Hoare triples used in the partial annotation. Since $\mathcal{A}_1$ has one state for each of the five assertions used in the partial annotation (namely *true*, $x \leq -1$, $x = -1$, *false* and $0 \leq x$), we can use the five program locations as automaton states and take the set of states $Q = \{\ell_0, \ldots, \ell_3, \ell_{\text{err}}\}$. Since $\mathcal{A}_1$ has one transition for each of the four Hoare triples, the transitions are exactly the four edges in the graph in Figure 6a.

We now proceed to define the partition of the executions of $\mathcal{P}_{\text{ex3}}$. We compute the program $\mathcal{P}^1_{\text{ex3}}$ as the intersection of the program $\mathcal{P}_{\text{ex3}}$ with the automaton $\mathcal{A}_1$ and the program $\mathcal{P}^2_{\text{ex3}}$ as the difference between $\mathcal{P}_{\text{ex3}}$ and $\mathcal{A}_1$.

$$\mathcal{P}^1_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \cap \mathcal{A}_1$$
$$\mathcal{P}^2_{\text{ex3}} = \mathcal{P}_{\text{ex3}} \setminus \mathcal{A}_1$$

We here exploit the fact that a program can be viewed as an automaton *and vice versa*. When we view a program as an automaton, we can apply set-theoretic operations (here intersection and set difference). When we view an automaton as a program, we can consider its operational semantics, apply a static analysis, check its correctness, and so on.

The executions of $\mathcal{P}^1_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have been proven correct by the annotation in Figure 6a, and the executions of $\mathcal{P}^2_{\text{ex3}}$ are exactly those executions of $\mathcal{P}_{\text{ex3}}$ that have not yet been proven correct.

In our example, $\mathcal{P}^1_{\text{ex3}}$ happens to be equal to $\mathcal{A}_1$ (since $\mathcal{A}_1$ is a subset of $\mathcal{P}_{\text{ex3}}$). We depict the program $\mathcal{P}^2_{\text{ex3}}$ in Figure 6b. The two programs capture the above-mentioned two cases of executions. For each of them, the application of interval analysis proves the correctness, i.e., the unreachability of the error location.

2 Programs, Correctness, and Automata

We first present an abstract formal setting in which we define the notions of: trace, correctness of a trace, program, and correctness of a program, in terms of automata-theoretic concepts. To help intutition, we then discuss how the setting relates to some of the more concrete settings that are commonly used.

2.1 Formal Setting

Trace τ. We assume a fixed set of statements Σ. A *trace* τ is a sequence of statements, i.e.,

$$\tau = \mathit{st}_1 \dots \mathit{st}_n$$

where $\mathit{st}_1, \dots, \mathit{st}_n \in \Sigma$ and $n \geq 0$ (the sequence is possibly empty). In order to connect our formal setting with automata theory, we view a statement st as a letter and a trace τ as a word over the alphabet Σ; i.e., $\tau \in \Sigma^\star$. Since one calls a set of words a *language*, the set of traces is the language of all words over the alphabet Σ.

$$\{\text{traces}\} = \Sigma^\star$$

Correctness of a trace τ. We assume a fixed set Φ of *assertions*. The set of assertions Φ contains the assertions *true* and *false* and comes with a binary relation, the *entailment* relation. We write $\varphi \models \psi$ if the assertion φ entails the assertion ψ.

We assume a fixed set of triples of the form $(\varphi, \mathit{st}, \psi)$ where φ and ψ are assertions in Φ and st is a statement in Σ. We say that every triple $(\varphi, \mathit{st}, \psi)$ in the set is a valid Hoare triple and we write

$$\{\varphi\}\ \mathit{st}\ \{\psi\} \quad \text{is valid}$$

(we abstract away from the procedure that establishes the validity of a Hoare triple).

The Hoare triple $\{\varphi\}\ \tau\ \{\psi\}$ is valid for the trace $\tau = \mathit{st}_1 \dots \mathit{st}_n$ if each of the Hoare triples below is valid, for some sequence of intermediate assertions $\varphi_1, \dots, \varphi_n$.

$$\{\varphi\}\ \mathit{st}_1\ \{\varphi_1\}, \dots, \{\varphi_{n-1}\}\ \mathit{st}\ \{\psi\}$$

If $n = 0$ and τ is the empty trace ($\tau = \varepsilon$) then φ must entail ψ.

We define the correctness of a trace wrt. a pair of assertions, the *pre/ postcondition pair*

$$(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}}).$$

The trace τ is defined to be correct if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid.

$$\{\text{correct traces}\} = \{\tau \in \Sigma^\star \mid \{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\} \text{ is valid}\}$$

The notion of a trace and the correctness of a trace are independent of a given program. We will next introduce the notion of a program and define the set of its *control flow traces*. We can then define the correctness of the program: the program is correct if all its control flow traces are correct.

Program. We formalize a program $\mathcal{P}$ as a special kind of graph which we call a *control flow graph*. The vertices of the control flow graph are called *locations*. The set of locations Loc contains a distinguished *initial* location ℓ_0 and a subset F of distinguished *final* locations. The edges of the control flow graph are labeled with statements. We use δ for the labeled edge relation; i.e.,

$$\delta \subseteq \mathsf{Loc} \times \Sigma \times \mathsf{Loc}.$$

The edge between the two locations ℓ and ℓ' is labeled by the statement st if δ contains the triple $(\ell, \mathit{st}, \ell')$.

Given a program $\mathcal{P}$, we say that the trace τ is a *control flow trace* if τ labels a path in the control flow graph between the initial location and a final location (the path need not be simple, i.e., it may repeat locations and edges).

Since a statement st is a letter of the alphabet Σ, the program

$$\mathcal{P} = (\mathsf{Loc}, \delta, \ell_0, F)$$

is an automaton over the alphabet Σ. Since a trace τ is a word (i.e., $\tau \in \Sigma^\star$), the automaton $\mathcal{P}$ recognizes a set of traces. We write $\mathcal{L}(\mathcal{P})$ for the language recognized by $\mathcal{P}$, which is a language of words over the alphabet Σ, i.e.,

$$\mathcal{L}(\mathcal{P}) \subseteq \Sigma^\star.$$

The condition that a trace τ is a control flow trace translates to the fact that the word τ is accepted by the automaton $\mathcal{P}$. Thus, the set of control flow traces is the language over the alphabet Σ which is recognized by $\mathcal{P}$, i.e.,

$$\{\text{control flow traces}\} = \mathcal{L}(\mathcal{P}).$$

Correctness of a program $\mathcal{P}$. We define that the program $\mathcal{P}$ is *correct* and write

$$\{\varphi_{\mathsf{pre}}\}\ \mathcal{P}\ \{\varphi_{\mathsf{post}}\}$$

if the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid for every control flow trace τ of $\mathcal{P}$, which is equivalent to the inclusion

$$\{\text{control flow traces}\} \subseteq \{\text{correct traces}\}.$$

Thus, the correctness of a program is characterized by the inclusion between two languages over the alphabet of statements Σ. The language of correct traces is in general not recognizable by a finite automaton; if, for example, the one-letter alphabet consisting of the statement `x++`, the precondition "`x` is equal to 0" and the postcondition "`x` is a prime number", then the language of correct traces is the set of traces whose length is a prime number. However, for every correct program $\mathcal{P}$ there exists a finite automaton $\mathcal{A}$ that *interpolates* between the language of control flow traces and the language of correct traces (i.e., $\mathcal{A}$ accepts all control flow tracs and $\mathcal{A}$ accepts only correct traces), formally

$$\{\text{control flow traces}\} \subseteq \mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}.$$

The existence of such an automaton $\mathcal{A}$ follows for a trivial reason. If we assume that the program $\mathcal{P}$ is correct, then we can choose $\mathcal{A}$ to be the program $\mathcal{P}$ itself (by the definition of control flow traces, $\mathcal{P}$ accepts all control flow traces, and by the definition of program correctness, $\mathcal{P}$ accepts only correct traces). In fact, $\mathcal{P}$ is the *smallest* among all automata that one can use to prove the correctness of the program $\mathcal{P}$. Many existing methods for proving program correctness are restricted to this one example as the choice for the automaton $\mathcal{A}$. In Section 3 we will discuss other examples (examples of automata $\mathcal{A}$ that properly interpolate between the language of control flow traces and the language of correct traces).

2.2 Discussion

We next relate the abstract formal setting used above to some of the more concrete settings that are commonly used.

Assertions. Usually, an assertion φ is a first-order logic formula over a given vocabulary. Its variables are taken from a set Var of variables (the *program variables*). In our formal setting, we will not introduce states and related notions (state predicate, postcondition, ...). In a formal setting based on states, an assertion is used to define a state predicate (i.e., a set of valuations or states). There, the Hoare triple $\{\varphi\}\ st\ \{\psi\}$ signifies that the postcondition of φ under the statement st entails ψ, or: if the statement st is executed in a state that satisfies φ then the successor state satisfies ψ. In our setting, we abstract away from the procedure (first-order theorem prover, SMT solver, ...) used to establish entailment or the validity of a Hoare triple. We will also abstract away from the specific procedure (static analysis, interpolant generation, ...) used to construct the sequence of intermediate assertions $\varphi_1, \ldots, \varphi_{n-1}$ for a given Hoare triple for a trace of the form above.

Using `assume` statements. In order to accommodate control constructs like `if-then-else` and `while` of programming languages in a formal setting based on the control flow graph, we can use a form of statement that is often called `assume` statement. That is, for every assertion ψ we have an statement (also written ψ) such that the Hoare triple $\{\varphi\}\ \psi\ \{\varphi'\}$ is valid if the assertion φ' is entailed by the conjunction $\psi \wedge \varphi$. The meaning of *assume* statements for the purpose

of verification is clear. Their operational meaning is somewhat contrived: if an execution reaches the statement ψ in a state that satisfies the assertion ψ then the statement is ignored, and if an execution reaches the statement ψ in a state that violates the assertion ψ then the execution is blocked (and the successor location in the control flow graph is not reached).

Infeasibility $\implies$ *Correctness.* Formally, we define that a trace is infeasible if the Hoare triple

$$\{\mathit{true}\}\ \tau\ \{\mathit{false}\}$$

is valid. Intuitively, a trace τ is infeasible if there is no possible execution of the sequence of the statements in τ (in whatever valuation of the program variables the execution starts, one of the `assume` statements in the sequence cannot be executed). For example, the sequence of two `assume` statements `x==0` `x==1` is an infeasible trace; the sequence `x:=0` `x==1` is another example. An infeasible trace thus satisfies every possible pre/postcondition pair. In other words, an infeasible trace is correct (for whatever pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ defining the correctness).

The fact that infeasibility implies correctness is crucial. In general, the set of *feasible* control flow traces is not regular (the feasible control flow traces are exactly the sequences of statements along paths in the *transition system* of the program, i.e., in the—in general infinite—graph formed by transitions between program states). We obtain a regular set because we include the infeasible control flow traces (in addition to the feasible ones). As an aside, if the program $\mathcal{P}$ is not correct then the set of correct control flow traces is in general not regular.

Non-reachability of error locations. In some settings, it is convenient to express the correctness of a program by the non-reachability of distinguished locations (often called *error locations*). In our setting, this corresponds to the special case where the set F consists of those locations and the postcondition φ_{post} is the assertion *false*. Moreover, if the precondition is *true*, the non-reachability of error locations is exactly the infeasibility of all control flow traces.

As mentioned above, an infeasible trace is correct (for any pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ that is used to define correctness of traces). In the special case of the pre/postcondition pair $(\mathit{true}, \mathit{false})$, the converse holds as well. I.e., in this case we have: a trace is correct exactly when it is infeasible.

Validity of `assert` *statements.* In other settings, it is convenient to express the correctness by the validity of *assert* statements. Informally, the statement `assert(e)` is valid if, whenever the statement is reached in an execution of the program, the Boolean expression e evaluates to *true*. This notion of correctness can be reduced to non-reachability (for each `assert` statement `assert(e)` for the expression `e`, one adds an edge to a new error location labeled with the assume statement `assume (not e)` for the negation of the expression `e`.)

Partial correctness. The partial correctness wrt. a general pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$ can always be reduced to the partical correctness wrt. the special case of the precondition being *true* and the postcondition being *false* (by modifying the control flow graph of the program: one adds an edge from a new initial location to the old one labeled with the assume statement $[\varphi_{\mathsf{pre}}]$ for the precondition and an edge from each old final location to a new final location (an "error location") labeled with the assume statement $[\neg\varphi_{\mathsf{post}}]$ for the negated postcondition).

3 Floyd-Hoare Automata

Given an automaton $\mathcal{A} = (Q, \delta, q_0, Q_{\mathsf{final}})$ over the alphabet of statements Σ and given a pre/postcondition pair $(\varphi_{\mathsf{pre}}, \varphi_{\mathsf{post}})$, when do we know that $\mathcal{A}$ accepts only correct traces (i.e., traces τ such that the Hoare triple $\{\varphi_{\mathsf{pre}}\}\ \tau\ \{\varphi_{\mathsf{post}}\}$ is valid)? The definition below gives a general condition (and provides a general method to construct an automaton that accepts only correct traces).

Definition 1. *The automaton $\mathcal{A} = (Q, \delta, q_0, Q_{\mathit{final}})$ over the alphabet of statements Σ (with the finite set of states Q, the transition relation $\delta \subseteq Q \times \Sigma \times Q$, the initial state q_0 and the set of final states Q_{final}) is a* Floyd-Hoare automaton *if there exists a mapping*

$$q \in Q \ \mapsto\ \varphi_q \in \Phi$$

that assigns to each state q an assertion φ_q such that

- *for every transition $(q, \mathit{st}, q') \in \delta$ from state q to state q' reading the letter st, the Hoare triple $\{\varphi_q\}\ \mathit{st}\ \{\varphi_{q'}\}$ is valid for the assertions φ_q and $\varphi_{q'}$ assigned to q and q', respectively,*
- *the precondition φ_{pre} entails the assertion assigned to the initial state,*
- *the assertion assigned to a final state entails the postcondition φ_{post}.*

$$\begin{aligned} (q, \mathit{st}, q') \in \delta &\implies \{\varphi_q\}\ \mathit{st}\ \{\varphi_{q'}\} \quad \textit{is valid} \\ q = q_0 &\implies \varphi_{\mathit{pre}} \models \varphi_q \\ q \in Q_{\mathit{final}} &\implies \varphi_q \models \varphi_{\mathit{post}} \end{aligned}$$

The mapping $q \mapsto \varphi_q$ from states to assertions in the definition above is called an *annotation* of the automaton $\mathcal{A}$.

Theorem 1. *A Floyd-Hoare automaton $\mathcal{A}$ accepts only correct traces,*

$$\mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}$$

i.e., if the trace τ is accepted by $\mathcal{A}$ then the Hoare triple $\{\varphi_{\mathit{pre}}\}\ \mathit{st}\ \{\varphi_{\mathit{post}}\}$ is valid.

Proof. We first prove, by induction over the length of the trace τ, the statement: for every pair of states q and q' and their corresponding assertions φ_q and $\varphi_{q'}$, if the trace τ labels some path from q to q' then the Hoare triple $\{\varphi_q\}\ \tau\ \{\varphi_{q'}\}$ is valid. The base case ($\tau = \varepsilon$ and $q = q'$) holds trivially. The induction step ($\tau' = \tau.\mathit{st}$ and $(q', \mathit{st}, q'') \in \delta$) uses the Hoare triple $\{\varphi_{q'}\}\ \mathit{st}\ \{\varphi_{q''}\}$ to show that $\{\varphi_q\}\ \tau.\mathit{st}\ \{\varphi_{q''}\}$ is valid. The theorem is the instance of the statement where we set q to the initial state and q' to a final state. □

A Floyd-Hoare automaton $\mathcal{A}$ is a correctness proof for the program $\mathcal{P}$ if it accepts all control flow traces, i.e., if $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$. Many (if not all) existing verification methods amount to constructing an annotation for the program $\mathcal{P}$ and thus to showing that $\mathcal{P}$ is a Floyd-Hoare automaton. Trivially, $\mathcal{P}$ can only be the smallest among all the Floyd-Hoare automata that prove the correctness of $\mathcal{P}$.

A well-known fact from the practice of automata is that the size of an automaton can be drastically reduced if the automaton is allowed to recognize a larger set. This is interesting in a setting where the size of the automaton we construct correlates with the number of Hoare triples we have to provide for an annotation. That is, instead of providing Hoare triples for an annotation of the control flow graph of the program $\mathcal{P}$ which recognizes exactly the set of control flow traces, it may be more efficient to provide Hoare triples for an annotation of the transition graph of an automaton $\mathcal{A}$ that recognizes a larger set (one can easily give examples of programs where an exponential reduction in proof size (when going from $\mathcal{P}$ to $\mathcal{A}$) can be obtained). If $\mathcal{A}$ is different from $\mathcal{P}$, one still needs to check that $\mathcal{A}$ is indeed a proof for $\mathcal{P}$, i.e., that the inclusion between the two automata, $\mathcal{L}(\mathcal{P}) \subseteq \mathcal{L}(\mathcal{A})$, does indeed hold. The efficiency of the proof check is thus an issue of efficient implementions of automata.

The next statements means that we can compose correctness proofs in the form of Floyd-Hoare automata to a correctness proof for the program $\mathcal{P}$.

Theorem 2. *If the Floyd-Hoare automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$ cover the set of control flow traces of the program $\mathcal{P}$ (i.e., $\mathcal{P} \subseteq \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$) then $\mathcal{P}$ is correct.*

Construction of a Floyd-Hoare automaton. The definition of Floyd-Hoare automata provides a general method to construct an automaton that accepts only correct traces, namely from a set of Hoare triples. We obtain different instances of the method by changing the approach to obtain the set of Hoare triples. For example, the set may stem from a *partial annotation* for the program, or the set of Hoare triples may be implicit from the *infeasibility proof* for a trace.

Let H be a finite set of Hoare triples, i.e., for each $(\varphi, \mathit{st}, \psi)$ in H, the Hoare triple $\{\varphi\}\ \mathit{st}\ \{\psi\}$ is valid (for now, we leave open how H is obtained). Let Φ_H be the finite set of assertions occuring in H. We assume that Φ_H includes the precondition φ_{pre} and the postcondition φ_{post}. We construct the Floyd-Hoare automaton $\mathcal{A}_H$ as follows.

$$\mathcal{A}_H = (Q_H, \delta, q_0, Q_{\mathsf{final}}) \quad \text{where} \quad \begin{aligned} Q_H &= \{q_\varphi \mid \varphi \in \Phi_H\} \\ \delta &= \{(q_\varphi, \mathit{st}, q_\psi) \mid (\varphi, \mathit{st}, \psi) \in H\} \\ q_0 &= q_{\varphi_{\mathsf{pre}}} \\ Q_{\mathsf{final}} &= \{q_{\varphi_{\mathsf{post}}}\} \end{aligned}$$

That is, we form the set of states Q_H by introducing a state q_φ for every assertion φ in H (i.e., Φ_H is bijective to Q_H). The transition relation δ defines a transition labeled by the letter st from the state q_φ to the state q_ψ for every Hoare triple $(\varphi, \mathit{st}, \psi)$ in H. The initial state is the state assigned to the precondition φ_{pre}. The (unique) final state is the state assigned to the postcondition φ_{post}.

Clearly, $\mathcal{A}_H$ is a Floyd-Hoare automaton: the inverse of the mapping $\varphi \mapsto q_\varphi$ is a mapping of states to assertions as required in Definition 1.

Construction of automaton from infeasibility proof. Assume, for example, that the trace $\tau = \mathit{st}_1 \ldots \mathit{st}_i \ldots \mathit{st}_j \ldots \mathit{st}_n$ is infeasible and that we have a proof of the form: the sequence of the two statements st_i and st_j is infeasible and the statements in between do not modify any variable used in st_i and st_j. We can construct an automaton with the set of states $Q = \{q_0, q_1, q_2\}$ as follows. The initial state q_0 has a transition to a state q_1 reading st_1. The state q_1 has a transition to the final state q_2 reading st_2. The initial state and the final state each have a self-loop standing for a transition reading any letter of the alphabet. The state q_1 has a self-loop standing for a transition reading any letter except for statements that modify a variable used in st_i or st_j. The construction generalizes to the case where the infeasibility involves more than two statements; see Example 1 in the introduction.

To see that this construction is a special case of the construction above, we take any assertion φ such that the two Hoare triples $\{\mathit{true}\}\ \mathit{st}_1\ \{\varphi\}$ and $\{\varphi\}\ \mathit{st}_2\ \{\mathit{false}\}$ are valid (for example, we can take the strongest postcondition of *true* under the statement st_1) and we form the set of Hoare triples H by those two Hoare triples, the Hoare triples of the form $\{\varphi\}\ \mathit{st}\ \{\varphi\}$ for any statement st in Σ that does not modify a variable used in st_i or st_j, and the trivial Hoare triples $\{\mathit{true}\}\ \mathit{st}\ \{\mathit{true}\}$ and $\{\mathit{false}\}\ \mathit{st}\ \{\mathit{false}\}$ for every statement st in Σ. Thus, the special case consists of leaving the assertion φ implicit. This is not always possible; see Example 2 in the introduction.

Construction of a correctness proof for the program $\mathcal{P}$. By Theorem 2, we can construct an automaton $\mathcal{A}$ for a correctness proof for the program $\mathcal{P}$, i.e.,

$$\{\text{control flow traces}\} \subseteq \mathcal{L}(\mathcal{A}) \subseteq \{\text{correct traces}\}. \tag{1}$$

as the union of Floyd-Hoare automata, i.e., $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$. The construction of $\mathcal{A}_1, \ldots, \mathcal{A}_n$ can be done in parallel from the n correctness proofs (i.e., infeasibility proofs or sets of Hoare triples) for some choice of traces $\tau_1, \ldots, \tau_n$. The construction of $\mathcal{A}$ as the union $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$ can also be done incrementally (for $n = 0, 1, 2, \ldots$) until (1) holds. Namely, if the inclusion does not yet hold (which is the case initially, when $n = 0$), then there exists a control flow trace τ_{n+1} which is not in $\mathcal{A}$. We then construct the automaton $\mathcal{A}_{n+1}$ from the proof for the trace τ_{n+1} and add it to the union, i.e., we update $\mathcal{A}$ to $\mathcal{A} \cup \mathcal{A}_{n+1}$. This is how we proceeded for the examples in the introduction.

If (1) holds for $\mathcal{A} = \mathcal{A}_1 \cup \ldots \cup \mathcal{A}_n$ then the programs $\mathcal{P}_1, \ldots, \mathcal{P}_n$ where $\mathcal{P}_i = \mathcal{P} \cap \mathcal{A}_i$ (for $i = 1, \ldots, n$) define a decomposition of the program $\mathcal{P}$ (i.e., $\mathcal{P} = \mathcal{P}_1 \cup \ldots \cup \mathcal{P}_n$). This decomposition is constructed automatically from correctness proofs (in contrast with an approach where one constructs correctness proofs for the modules of a given decomposition).

```
ℓ0: if(nondet){
        x:=0
    } else {
        y:=0
    }
ℓ1: if(z==0) {
ℓ2:     assert(z==0)
    } else {
ℓ3:     assert(x==0 || y==0)
    }
```

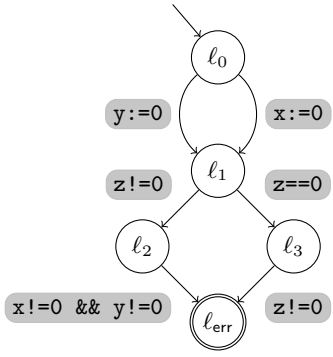


Fig. 7. Example program where it is useful to remove the restriction that the correctness argument must be based on (an unfolding of) the control flow graph

4 Conclusion and Future Work

We have presented a new angle of attack at the problem of finding the right abstraction of a program for a given correctness property. Existing approaches (see our list of references) differ mainly in their techniques to decompose the problem. Often the techniques (unfolding, splitting nodes, abstract states, ...) are based on the control flow graph. Phrased in the terminology of our setting, the techniques amount to constructing a cover (often a partitioning) of the set of control flow traces by automata $\mathcal{A}_1, \ldots, \mathcal{A}_n$. The construction is restricted in that the automata must be merged into one automaton and, moreover, the states and transitions of the resulting automaton must be in direct correspondence with the nodes and edges of the control flow graph. This is needed to ensure that all control flow traces are indeed covered (in the absense of an inclusion check). Our approach allows one to remove this restriction. The example in Figure 7 may be used to illustrate the difference between the approaches.

It is a topic of future work to position existing approaches to software model checking in our setting. Since an *abstraction refinement* step eliminates in general not just one counterexample trace but a whole set, it may be interesting to characterize this set by an automaton and thus quantify the *progress* property.

The setting presented here can be extended to automata over *nested words* in order to account for programs with (possibly recursive) procedures [12], and to *alternating automata* in order to account for concurrent programs [9]. It is still open how one can extend the setting to *Büchi automata* in order to account for termination and cost analysis, although the use of omega-regular expressions to decompose the set of infinite traces in [10] may be a step in this direction.

The development of a practical method based on Floyd-Hoare automata must address a wide range of design choices. An initial implementation is part of ongoing work [11].

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 158–172. Springer, Heidelberg (2002)
2. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309. ACM (2007)
3. Bradley, M., Cassez, F., Fehnker, A., Given-Wilson, T., Huuck, R.: High performance static analysis for industry. ENTCS 289, 3–14 (2012)
4. Brückner, I., Dräger, K., Finkbeiner, B., Wehrheim, H.: Slicing abstractions. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 17–32. Springer, Heidelberg (2007)
5. Christ, J., Hoenicke, J., Nutz, A.: Proof tree preserving interpolation. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 124–138. Springer, Heidelberg (2013)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252. ACM (1977)
8. Cousot, P., Ganty, P., Raskin, J.-F.: Fixpoint-guided abstraction refinements. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 333–348. Springer, Heidelberg (2007)
9. Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. In: POPL, pp. 129–142 (2013)
10. Gulwani, S., Jain, S., Koskinen, E.: Control-flow refinement and progress invariants for bound analysis. In: PLDI, pp. 375–385 (2009)
11. Heizmann, M., Christ, J., Dietsch, D., Ermis, E., Hoenicke, J., Lindenmann, M., Nutz, A., Schilling, C., Podelski, A.: Ultimate Automizer with SMTInterpol (competition contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 641–643. Springer, Heidelberg (2013)
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL, pp. 471–482 (2010)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM (2002)
14. Jha, S.K., Krogh, B.H., Weimer, J.E., Clarke, E.M.: Reachability for linear hybrid automata using iterative relaxation abstraction. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 287–300. Springer, Heidelberg (2007)
15. Junker, M., Huuck, R., Fehnker, A., Knapp, A.: SMT-based false positive elimination in static program analysis. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 316–331. Springer, Heidelberg (2012)
16. Kroening, D., Weissenbacher, G.: Interpolation-based software verification with Wolverine. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 573–578. Springer, Heidelberg (2011)
17. Long, Z., Calin, G., Majumdar, R., Meyer, R.: Language-theoretic abstraction refinement. In: de Lara, J., Zisman, A. (eds.) FASE 2012. LNCS, vol. 7212, pp. 362–376. Springer, Heidelberg (2012)

18. Mauborgne, L., Rival, X.: Trace partitioning in abstract interpretation based static analyzers. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 5–20. Springer, Heidelberg (2005)
19. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
20. Segelken, M.: Abstraction and counterexample-guided construction of *omega* - automata for model checking of step-discrete linear hybrid models. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 433–448. Springer, Heidelberg (2007)
21. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 332–344. IEEE Computer Society (1986)

Multi-solver Support in Symbolic Execution

Hristina Palikareva and Cristian Cadar

Department of Computing, Imperial College London
London, United Kingdom
{h.palikareva,c.cadar}@imperial.ac.uk

Abstract. One of the main challenges of dynamic symbolic execution—an automated program analysis technique which has been successfully employed to test a variety of software—is constraint solving. A key decision in the design of a symbolic execution tool is the choice of a constraint solver. While different solvers have different strengths, for most queries, it is not possible to tell in advance which solver will perform better.

In this paper, we argue that symbolic execution tools can, and should, make use of multiple constraint solvers. These solvers can be run competitively in parallel, with the symbolic execution engine using the result from the best-performing solver.

We present empirical data obtained by running the symbolic execution engine `KLEE` on a set of real programs, and use it to highlight several important characteristics of the constraint solving queries generated during symbolic execution. In particular, we show the importance of constraint caching and counterexample values on the (relative) performance of `KLEE` configured to use different SMT solvers.

We have implemented multi-solver support in `KLEE`, using the `metaSMT` framework, and explored how different state-of-the-art solvers compare on a large set of constraint-solving queries. We also report on our ongoing experience building a parallel portfolio solver in `KLEE`.

1 Introduction

Symbolic execution [14] is a program analysis technique that can systematically explore paths through a program by reasoning about the feasibility of explored paths using a constraint solver. The technique has gathered significant attention in the last few years [6], being implemented in several tools, which have found deep bugs and security vulnerabilities in a variety of software applications [4].

One of the key factors responsible for the recent success of symbolic execution techniques are the recent advances in constraint-solving technology [10]. Nevertheless, constraint solving remains one of the main challenges of symbolic execution, and for many programs it is the main performance bottleneck. As a result, a key decision when designing a symbolic execution tool is the choice of a constraint solver. While this choice may be affected by the solver's supported theories, specific optimisations or software licenses, in many cases it is somewhat arbitrary, and it is not always clear which solver is the best match for a given symbolic execution tool. In fact, given two state-of-the-art solvers, it is unlikely

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 53–68, 2013.

that one consistently outperforms the other; more likely, one solver is better on some benchmarks and worse on others. Moreover, for a given query, it is often not possible to tell in advance which solver would perform better.

In this paper, we argue that symbolic execution tools can—and should—make use of multiple constraint solvers. These solvers can be run competitively in parallel, with the symbolic execution engine using the result from the best-performing solver. We believe such an approach is particularly timely in the age of parallel hardware platforms, such as multi-core CPUs [5].

The idea of using a *portfolio* of solvers is not new: this technique was already employed in the context of SAT solving [13,24], SMT solving [23], and bounded model checking [9], among others. However, as far as we know, this is the first paper that reports on how different SMT solvers compare in the context of symbolic execution.

The main contributions of this paper are:

1. A discussion of the main characteristics of the constraint-solving queries generated in symbolic execution, accompanied by detailed statistics obtained from real symbolic execution runs;
2. An analysis of the effect of constraint caching and counterexample values on the performance of symbolic execution;
3. A comparison of several state-of-the-art SMT solvers for closed quantifier-free formulas over the theory of bitvectors and bitvector arrays (`QF_ABV`) on queries obtained during the symbolic execution of real-world software;
4. An extension of the popular symbolic execution engine `KLEE` [2] that supports multiple SMT solvers, based on the `metaSMT` [12] solver framework.
5. A discussion of our ongoing experience building a portfolio solver in `KLEE`.

The rest of the paper is organised as follows. Section 2 provides background information on symbolic execution and the `KLEE` system. Section 3 presents the `metaSMT` framework and its integration with `KLEE`. Then, Section 4 analyses the constraint-solving queries obtained during the symbolic execution of several real applications, and discusses how different solvers perform on these queries. Finally, Section 5 discusses how symbolic execution could benefit from a parallel portfolio solver, Section 6 presents related work and Section 7 concludes.

2 Background

Dynamic symbolic execution is a program analysis technique whose goal is to systematically explore paths through a program, reasoning about the feasibility of each explored path using a Satisfiability Modulo Theory (SMT) constraint solver. In addition, symbolic execution systematically checks each explored path for the presence of generic errors such as buffer overflows and assertion violations.

At a high level, the program is executed on a *symbolic* input, which is initially unconstrained. For example, in the code in Figure 1, the symbolic input is the integer variable x, which is in the beginning allowed to take any value. Then, as the program executes, each statement that depends on the symbolic input adds

```
1  int main() {
2      unsigned a[5] = {0, 1, 1, 0, 0};
3      unsigned x = symbolic();
4      unsigned y = x+1;
5      if (y < 5) {
6          if (a[y])
7              printf("Yes\n");
8          else printf("No\n");
9      }
10     else printf("Out of range\n");
11     return 0;
12 }
```

Fig. 1. Code example illustrating some of the main aspects of symbolic execution

further constraints on the input. For instance, the statement `y=x+1` on line 4 constrains y to be equal to $x + 1$. When a branch that depends on the symbolic input is reached, if both branch directions are feasible, symbolic execution follows them both, constraining the branch condition to be *true* on the true path, and *false* on the other. In our example, when the program reaches the branch on line 5, execution is forked into two paths: on the `then` path, the constraint $y < 5$ is added and execution proceeds to line 6, while on the `else` path, the constraint $y \geq 5$ is added and execution proceeds to line 10. However, note that constraints are added not in terms of intermediate variables such as y, but in terms of the initial symbolic inputs. That is, in our example, the constraints being added on each path are $x + 1 < 5$ and $x + 1 \geq 5$. There is one case in which constraints cannot solely be expressed in terms of the original inputs. This happens when a concrete array is indexed by a symbolic variable. In our example, the concrete array a is indexed by the symbolic variable y on line 6. In order to reason about the symbolic access $a[y]$, the constraint solver needs to know all the values of the array a. With this knowledge, the solver can determine that the branch at line 6 can be both *true* (when x is 0 or 1) and *false* (when x is 2 or 3).

Finally, when a path ends or an error is discovered, one can take all the constraints gathered along that path and ask the constraint solver for a concrete solution. This solution, also called a *counterexample*, represents a test case that exercises the path. In the context of software testing, these test cases can be used to form high-coverage test suites, as well as to generate bug reports.

2.1 Constraint Solving in Symbolic Execution

In this section, we discuss some of the most important characteristics of the constraint-solving queries generated during symbolic execution:

Large Number of Queries. This is perhaps the most important characteristic of constraint solving in symbolic execution. Unlike other constraint-based program testing and verification techniques that generate a small number of queries,[1] on a typical run, symbolic execution generates queries at every

[1] For instance, in bounded model checking [7] all paths up to a particular length are encoded as a single query.

symbolic branch and every potentially-dangerous symbolic operation it encounters, amounting to a large number of overall queries. As a result, in order to efficiently explore the program space, these queries need to be solved quickly, at a rate of tens or even thousands of queries per second.

Concrete Solutions. Symbolic execution often requires concrete solutions for satisfiable queries. These are needed to create test cases, interact with the outside world (e.g. before calling an external function, all symbolic bytes need to be replaced by concrete values), simplify constraints (e.g., double-pointer dereferences [3]), and reuse query results (e.g., `KLEE`'s counterexample cache [2]).

Array Operations. Arrays play an important role in symbolic execution. Many programs take as inputs arrays (in one form or another, e.g., strings are essentially arrays of characters), and concrete arrays often become part of the symbolic constraints when they are indexed by a symbolic input, as we have shown above. Furthermore, pointer operations in low-level code are also modelled using arrays. As a result, efficiently reasoning about arrays is extremely important in symbolic execution [3].

Bit-Level Accuracy. Many programs require bit-level accurate constraints, in order to reason about arithmetic overflow, bitwise operations, or integer and pointer casting. In particular, an important characteristic of the `KLEE` tool analysed in this paper is that it generates queries with bit-level accuracy.

2.2 Constraint Solving in KLEE

The experiments presented in this paper use `KLEE` [2], a modern symbolic execution engine available as open source from `http://klee.llvm.org`. `KLEE` works at the level of `LLVM` bitcode [15] and uses the constraint solver `STP` [11]. We now elaborate on some key issues related to constraint solving in `KLEE`.

The queries issued by `KLEE` are of two main types: *branch* and *counterexample queries*. Branch queries are issued when `KLEE` reaches a symbolic branch, to decide whether to follow only the *then*, only the *else*, or both sides of the branch. They are of the form (C, E) where C represents the set of constraints that hold on the current path (the *path constraints*), and E is a branch expression whose validity `KLEE` tries to establish. The possible answers are *provably true*, *provably false*, and *neither* (i.e., under the current set of constraints C, E could be both *true* and *false*). Branch queries are broken down into one or two satisfiability (or validity[2]) queries. For instance, to conclude that E is neither provably true nor provably false, `KLEE` needs to determine that both $\neg E$ and E are satisfiable.

Counterexample queries are used to request a solution for the current path constraints, e.g. when `KLEE` needs to generate a test case at the end of a program path. In addition, the counterexample cache in `KLEE` (described below) asks for a counterexample for all queries that are found to be satisfiable.

Before invoking `STP`, `KLEE` performs a series of constraint solving optimisations, which exploit the characteristics of the queries generated during symbolic

[2] In our context, a satisfiability query can be transformed into a validity query and vice-versa: a formula F is satisfiable iff $\neg F$ is not valid.

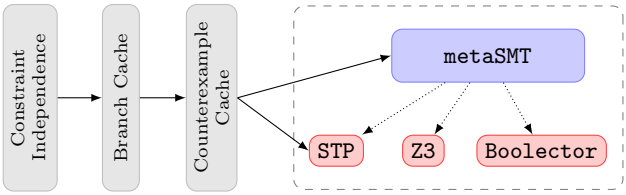


Fig. 2. Solver passes in KLEE, including the new metaSMT pass

execution. These optimisations are structured in KLEE as a sequence of solver passes, which are depicted in Figure 2; most of these passes can be enabled and disabled via KLEE's command-line options.

One of the key solver passes in KLEE is the elimination of redundant constraints, which we call *constraint independence* [3]. Given a branch query (C, E), this pass eliminates from C all the constraints which are not (transitively) related to E. For example, given $C = \{x < 10, z < 20, x + y = 10, w = 2z, y > 5\}$ and $E = x > 3$, this pass eliminates from C the constraints $z < 20$ and $w = 2z$, which don't influence E.

The other key solver passes are concerned with caching. KLEE uses two different caches: a branch cache, and a counterexample cache. The branch cache simply remembers the result of branch queries. The counterexample cache [2] works at the level of satisfying assignments. Essentially, it maps constraint sets to either a counterexample if the constraint set is satisfiable, or to a special sentinel if it is unsatisfiable. Then, it uses subset and superset relations among constraint sets to determine the satisfiability of previously unseen queries. For example, if the cache stores the mapping $\{x > 3, y > 2, x + y = 10\} \rightarrow \{x = 4, y = 6\}$, then it can quickly determine that the subset $\{x > 3, x + y = 10\}$ of the initial constraint set is also satisfiable, because removing constraints from a set does not invalidate any solutions. Similarly, the counterexample cache can determine that if a constraint set is unsatisfiable, any of its supersets are unsatisfiable too.

An important optimisation of the counterexample cache is based on the observation that many constraint sets are in a subset/superset relation. For example, as we explore a particular path, we always add constraints to the current path constraints. Furthermore, we observed that many times, if a subset of a constraint set has a solution, then often this solution holds in the original set too [2]. That is, adding constraints often does not invalidate an existing solution, and checking whether this holds (by simply substituting the solution in the constraints) is usually significantly cheaper compared to invoking the constraint solver. For example, if the cache stores the mapping $\{x > 3, y > 2, x+y = 10\} \rightarrow \{x = 4, y = 6\}$, then it can quickly verify that the solution $\{x = 4, y = 6\}$ also satisfies the superset $\{x > 3, y > 2, x+y = 10, x < y\}$ of the original constraint set, thus saving one potentially-expensive solver call. Given a constraint set, the counterexample cache tries all of its stored subsets, until it finds a solution (if any exists).

An important observation is that the cache hit rate depends on the actual counterexamples stored in the cache. For example, if instead the cache stored the mapping $\{x > 3, y > 2, x + y = 10\} \rightarrow \{x = 7, y = 3\}$, it would not be able to prove that the superset $\{x > 3, y > 2, x + y = 10, x < y\}$ is also satisfiable, since the cached assignment $\{x = 7, y = 3\}$ is not a solution for the superset.

3 The metaSMT Framework

One of the factors that has contributed to the recent progress in constraint-solving technology is the development of SMT-LIB [18], a common set of standards and benchmarks for SMT solvers. In particular, SMT-LIB defines a common language for the most popular SMT logics, including the fragment of *closed quantifier-free formulas over the theory of bitvectors and bitvector arrays* (`QF_ABV`), which is used by `KLEE`. Unfortunately, communicating via the textual constraint representation offered by the SMT-LIB format is not a feasible option in practice. The overhead of having the symbolic execution engine output SMT-LIB constraints and the solver parsing them back would be excessive. For example, we measured for a few benchmarks the average size of a single `KLEE` query in SMT-LIB format, and we found it to be on the order of hundreds of kilobytes, which would add significant overhead given the high query rate in symbolic execution (see §4).

As a result, it is critical to interact with solvers via their native APIs, and in our work we do so by using the `metaSMT` framework. `metaSMT` [12] provides a unified API for transparently using a number of SMT (and SAT) solvers, and offers full support for the `QF_ABV` logic used by `KLEE`. The unified C++ API provided by `metaSMT` is efficiently translated at compile time, through template meta-programming, into the native APIs provided by the SMT solvers. As a result, the overhead introduced by `metaSMT` is small, as we discuss in Section 4.

The solvers supported by `metaSMT` for the `QF_ABV` fragment are `Boolector` [1] and `Z3` [17]. `STP` only had support for `QF_BV`, and we extended it to fully handle the `QF_ABV` fragment. We contributed back our code to the `metaSMT` developers.

In `KLEE`, we added support for using the `metaSMT` API by implementing a new core solver pass, as depicted in Figure 2.

4 Experimental Evaluation

We evaluated our multi-solver `KLEE` extension on 12 applications from the `GNU Coreutils 6.10` application suite, which we used in prior work [2,16]. We selected only 12 out of the 89 applications in the `Coreutils` suite, in order to have time to run them in many different configurations (our experiments currently take days to execute). Our selection was unbiased: we first discarded all applications for which either (a) our version of `KLEE` ran into unsupported `LLVM` instructions or system calls,[3] (b) `KLEE` finished in less than one hour (e.g. `false`), or c) the

[3] Currently, `KLEE` does not fully support the `LLVM` 2.9 instruction set nor certain system calls in recent versions of Linux.

symbolic execution runs exhibited a significant amount of nondeterminism (e.g. for `date`, which depends on the current time, or for `kill`, where we observed very different instructions executed across runs). Out of the remaining applications, we selected the first 12 in alphabetical order.

We used `LLVM` 2.9 and `metaSMT` 3 in our experiments, and the SMT solvers `Boolector v1.5.118`, `Z3 v4.1` and `STP 32:1668M`, for which we used the default configuration options as provided by `metaSMT` (see also the threats to validity in §4.3). We configured `KLEE` to use a per-query timeout of 30s and 2 GB of memory. We ran all of our experiments on two similar Dell PowerEdge R210 II machines with 3.50 GHz Intel Xeon quad-core processors and 16 GB of RAM.

4.1 Solver Comparison Using the DFS Strategy and No Caching

For the first set of experiments, we ran each benchmark for one hour using `KLEE`'s default `STP` solver and recorded the number of executed `LLVM` instructions. In the following experiments, we ran each benchmark with `KLEE` for the previously recorded number of instructions, configured to use four different solver configurations: default `STP`, `metaSMT` with `STP`, with `Boolector` and with `Z3`.[4]

The main challenge is to configure `KLEE` to behave deterministically across runs; this is very difficult to accomplish, given that `KLEE` relies on timeouts, time-sensitive search heuristics, concrete memory addresses (e.g. values returned by `malloc`), and counterexample values from the constraint solver. To make `KLEE` behave as deterministically as possible, we used the depth-first search (DFS) strategy, turned off address-space layout randomisation, and implemented a deterministic memory allocator to be used by the program under testing. With this configuration, we have observed mostly deterministic runs with respect to the sequence of instructions executed and queries issued by `KLEE`—in particular, for all 12 benchmark considered, we observed that `KLEE`'s behaviour is very similar across `metaSMT` runs: e.g. modulo timeouts, `KLEE` consistently executed the same number of instructions with different solvers, and the number of queries only rarely differed by a very small number. We believe that for these benchmarks, the effect of any remaining nondeterminism is small enough to allow for a meaningful comparison.

MetaSMT Overhead. To measure the overhead introduced by `metaSMT`, we compared `KLEE` using directly the `STP` API with `KLEE` using `STP` via the `metaSMT` API. For 9 out of the 12 benchmarks, the overhead was small, at under 3%. For `ln` it was 6.7%. For `chmod` and `csplit`, the overhead was substantial (72.6% and 42.0% respectively), but we believe this is mainly due to the way `STP` expressions are exactly constructed via the two APIs.

Statistics for KLEE with STP. Table 1 presents some statistics about the runs with `KLEE` using its default solver `STP`, invoked via the `metaSMT` API.

[4] Note that re-running `KLEE` with `STP` for the same number of instructions does not always take one hour; this is due to the fact that on exit `KLEE` performs different activities than when run with a timeout. However, the goal of this initial run is only to obtain a fixed number of instructions for which to re-run each benchmark.

Table 1. Statistics for the DFS runs without caching, using STP via the metaSMT API: the instructions per second rate, the number of queries, the average query size, the queries per second rate in KLEE and STP respectively, and the percentage of time spent in all constraint-solving activities and STP respectively

Application	Instrs/sec	Queries	Q-size	Queries/sec		Solver(%)	
				total	STP	total	STP
[	3,914	197,282	2,868	55.1	60.0	97.8	89.8
base64	18,840	254,645	546	73.8	76.6	97.0	93.4
chmod	12,060	202,855	7,125	36.4	40.2	97.2	87.9
comm	73,064	586,485	120	189.0	201.9	88.4	82.7
csplit	10,682	244,803	2,179	49.7	52.7	98.3	92.7
dircolors	8,090	175,531	1,588	49.3	50.5	98.6	96.4
echo	227	114,830	6,852	34.8	41.7	98.8	82.3
env	21,955	379,421	664	109.1	119.8	97.2	88.5
factor	1,897	19,055	2,213	5.3	5.3	99.7	99.4
join	12,649	131,947	1,391	36.6	37.2	98.1	96.3
ln	13,420	366,926	786	103.8	115.3	97.0	87.4
mkfifo	25,331	221,308	2,144	62.3	67.4	96.6	89.3

The number of LLVM instructions executed per second varied between 227 for echo and 73,064 for comm, with a median of 12,355 instructions per second. The number of queries issued by each benchmark to the solver ranges between 19,055 for factor and 586,485 for comm, with a median of 212,082. The Queries/sec column shows two query rates for each benchmark: the first is the overall rate, i.e., number of queries over the total time spent by KLEE in constraint-solving activities (i.e. all activities in Figure 2), while the second is the rate seen by STP, i.e., the number of queries over the total time spent in STP.[5] The overall rate varies between 5.3 queries per second for factor and 189.0 for comm, with the median at 52.4 queries per second, while the STP rate varies between 5.3 queries per second for factor and 201.9 for comm, with the median at 56.4 queries per second. This is a high query rate; as we discussed in Section 2, constraint solving in symbolic execution has to handle a high number of queries per unit of time.

The column Q-size measures the average query size in terms of number of expression nodes, where shared expressions are only counted once. The set of the four smallest Q-size values corresponds to that of the four largest queries per second rates, but we did not observe any correlation for the other benchmarks.

The last column of Table 1 shows the percentage of time spent in constraint solving: the first number shows the percentage of time spent by KLEE in all constraint-solving activities, while the second number shows the percentage of time spent in STP. The key observation is that with the exception of comm, KLEE spends over 96% of the overall time in constraint-solving activities, with over 82% of the overall time spent in STP. This shows that for these benchmarks,

[5] STP times include the overhead incurred by using metaSMT.

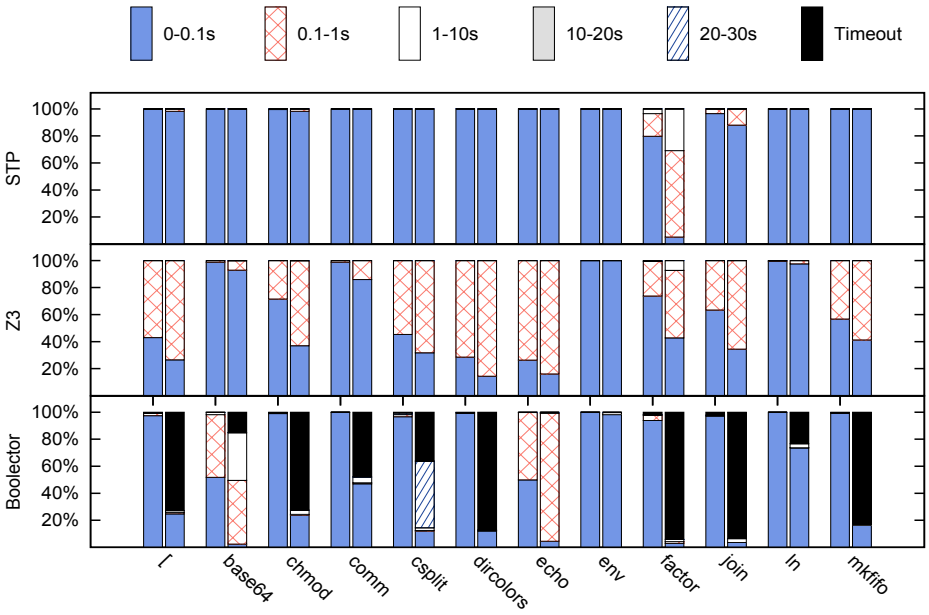


Fig. 3. Distribution of query types for the DFS runs without caching. For each benchmark, the left bar shows the percentage of queries that are processed by the core solver (via `metaSMT`) in 0–0.1s, 0.1–1s, 1–10s, 10–20s, 20–30s or reach the 30s timeout, while the right bar shows the percentage of time spent executing queries of each type.

constraint solving is the main performance bottleneck, and therefore optimising this aspect would have a significant impact on `KLEE`'s overall performance.

The upper chart in Figure 3 shows the distribution of query types for `KLEE` with `STP`. There are two bars for each benchmark: one showing the percentage of queries that finish in 0–0.1s, 0.1–1s, 1–10s, 10–20s, 20–30s, and time out; and one showing the percentage of time spent executing queries of each type. With the exception of `factor` and `join`, almost all queries (over 99%) take less than 0.1 seconds to complete, and `STP` spends almost all of its time (over 98%) solving such cheap queries. For `factor`, almost 80% of the queries still take less than 0.1s, but they account for only around 5% of the time.

Solver Comparison. To compare the performance of different solvers, we ran `KLEE` via `metaSMT` with `STP`, `Z3` and `Boolector` for a fixed number of instructions, as discussed above. Besides the 30s timeout set for each query, we also set an overall timeout of three hours for each run. `STP` and `Z3` have no query timeouts, while `Boolector` has query timeouts on all benchmarks with the exception of `echo` and `env`. Note that per-query timeouts may have a significant impact on the instructions executed by `KLEE`, since on a query timeout, we terminate the current execution path (which may have later spawned a large number of paths), and follow alternative parts of the execution tree instead.

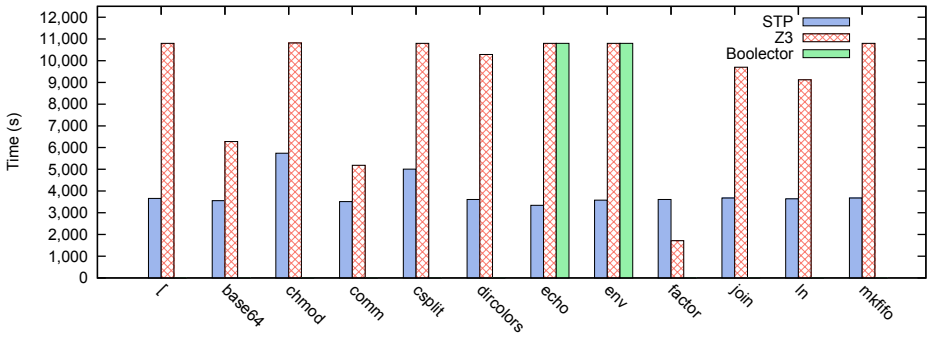


Fig. 4. Time taken by `KLEE` with `metaSMT` for `STP`, `Z3` and `Boolector`, using DFS and no caches. We set a timeout of 30s per query and an overall timeout of 10,800s (3h) per run. `Boolector` had query timeouts on all applications apart from `echo` and `env` and always reached the overall timeout except for `factor`. Query timeouts affect the subsequent queries issued by `KLEE`, which is why we only show two bars for `Boolector`.

Figure 4 shows the results. `STP` emerges as the clear winner: it is beaten on a single benchmark, and it has no query timeouts nor overall timeouts. The one notable exception is `factor`, where `Z3` finishes more than twice faster than `STP` (1,713s for `Z3` vs. 3,609s for `STP`). This categorical win for `STP` is not that surprising: `STP` was initially designed specifically for our symbolic execution engine `EXE` [3], and its optimisations were driven by the constraints generated by `EXE`. As a re-design of `EXE`, `KLEE` generates the same types of queries as `EXE`, and thus benefits from `STP`'s optimisations.

Figure 3 presents the distribution of query types for all solvers. Note that because of timeouts, the three solvers do not always execute the same queries. So although a precise comparison across solvers is not possible, this figure does show the mix of queries processed by each solver. While `STP` solves most of its queries in under 0.1s, the situation is different for the other two solvers: `Z3` often spends a lot of its time processing queries in the 0.1–1s range, while `Boolector` processes a wider variety of query types. In particular, it is interesting to note that while `Boolector` often spends most of its time in queries that time out, the percentage of these queries is quite small: for example, for `dircolors` there are only 0.5% queries that time out, but these consume 87.3% of the time. This illustrates the effect of the timeout value on the overall performance of symbolic execution: one may observe that decreasing the timeout value to 10s would not change the results for `STP` and `Z3`, but would help `Boolector` solve more queries.

4.2 Solver Comparison Using the DFS Strategy and Caching

We repeated the same experiment with `KLEE`'s caches enabled, to understand the effect of caching on solver time. That is, we ran `KLEE` using the `STP` API for one hour per benchmark, and we recorded the number of instructions executed in each case. We then re-ran `KLEE` on each benchmark for the previously recorded number of instructions using `STP`, `Z3` and `Boolector` via the `metaSMT` API.

Table 2. Statistics for the DFS runs with caching, using `STP` via the `metaSMT` API: the instructions per second rate, the number of queries and the queries per second rate in `KLEE` and `STP` respectively, and the percentage of time spent in all constraint-solving activities and `STP` respectively

Application	Instrs/sec	Queries		Queries/sec		Solver(%)	
		total	STP	total	STP	total	STP
[	695	30,838	30,613	7.9	58.3	99.6	13.4
base64	20,520	184,348	47,600	42.2	42.6	98.7	25.3
chmod	5,360	46,438	37,911	12.6	75.7	99.2	13.5
comm	222,113	1,019,973	21,720	305.0	83.5	87.9	6.8
csplit	19,132	285,655	33,623	63.5	28.0	98.1	26.2
dircolors	1,091,795	5,609,093	2,077	4,251.7	64.0	36.3	0.9
echo	52	16,318	764	4.5	52.7	99.7	0.4
env	13,246	96,425	38,047	26.3	63.8	98.5	16.1
factor	12,119	80,975	6,189	22.6	1.8	99.1	97.6
join	1,033,022	5,362,587	4,963	3,401.2	34.2	43.9	4.0
ln	2,986	91,812	40,868	24.5	62.7	99.4	17.3
mkfifo	3,895	26,631	25,622	7.2	58.1	99.3	11.9

We begin again by showing, in Table 2, some statistics about the runs with `KLEE` using `STP` via the `metaSMT` API. The table presents the same information as in Table 1, except that we omit the Q-size metric, and we show two numbers under the Queries column: the first is the number of queries issued by `KLEE` to the branch cache,[6] while the second is the number of queries issued to `STP` (i.e. those that miss in both caches).

First of all, it is interesting to compare the rate of instructions and queries issued by `KLEE` with and without caching. While the actual instructions executed in each case may be quite different, Tables 1 and 2 clearly show that caching sometimes helps significantly, while sometimes hurts performance. For example, `dircolors` goes from a relatively modest 8,090 instructions per second without caching to 1,091,795 with caching, and from only 49.3 queries per second without caching to 4,251.7 with caching. At the other end of the spectrum, `mkfifo` decreases its processing rate from 25,331 to 3,895 instructions per second, and from 62.3 to 7.2 queries per second. This illustrates the need for better, more adaptive caching algorithms.

The Solver column shows that `KLEE` still spends most of its time in constraint-solving activities (which include the caching code): for 9 out of the 12 benchmarks, `KLEE` consumes more than 98% of the time solving queries. On the other hand, the amount of time spent in `STP` itself decreases substantially compared to the runs without caching, with a median value at only 13.5%.

[6] Recall that the branch queries stored in this cache sometimes summarise the result of two solver queries.

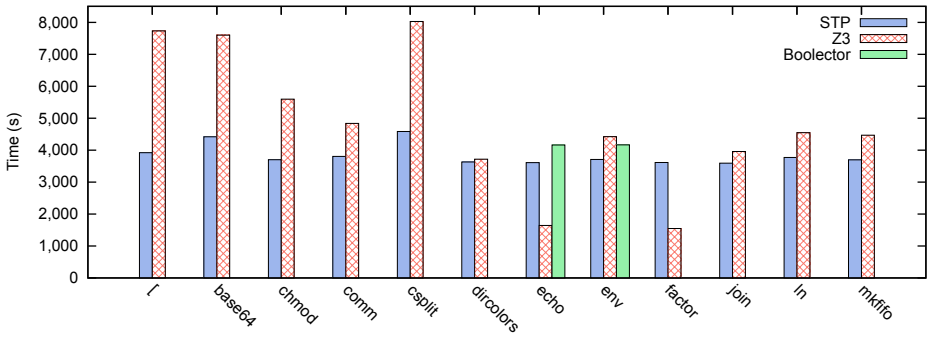


Fig. 5. Execution times for the DFS runs with caching. We set a timeout of 30s per query and an overall timeout of 10,800s (3h) per run. `Boolector` had individual query timeouts on all applications apart from `echo` and `env`. Query timeouts affect the subsequent queries issued by `KLEE`, which is why we only show two bars for `Boolector`.

Figure 5 shows the time taken by each solver on the DFS runs with caching. Overall, `STP` is still the clear winner, but the interesting fact is that caching can sometimes affect solvers in different ways, changing their relative performance. One such example is `echo`, where `Z3` wins: none of the solvers have timeouts, and all issue 16,318 overall queries, 764 of which reach the solver. However, `KLEE` with `Z3` spends significantly less time in the caching code, which accounts for its overall performance gain (interestingly enough, `STP` is faster than `Z3` for the 764 queries that reach the solver: 14s for `STP` vs. 74s for `Z3`). The widely different behaviour of the caching module is due to the different solutions returned by each solver: we noticed that with `Z3`, the counterexample cache tries significantly fewer subsets in its search for a solution.

Another interesting example is `env`: with all solvers, `KLEE` issues 96,425 queries. However, with `STP` and `Boolector`, only 38,047 reach the solver, while with `Z3`, 38,183 do. The 136 queries saved by the `STP` and `Boolector` runs are due to the fact that they were luckier with the solutions that they returned, as these solutions led to more hits afterwards.

To sum up, we believe these experiments demonstrate that the (relative) performance of `KLEE` running with different solvers can be affected, sometime substantially, by caching. Caching can significantly improve performance, but can also deteriorate it, so future work should focus on better, more adaptive caching algorithms. A key factor that affects caching behaviour are the solutions returned by the solver: first, different solutions can lead to different cache hit rates (as shown by the `env` runs), and second, even when the cache hit rate remains unchanged, different solutions can drastically affect the performance of the caching module, as demonstrated by the `echo` runs. Therefore, one interesting area of future work is to better understand the effect of the solutions returned by the solver on caching behaviour. In fact, we believe that this is also an aspect that could play an important role in the design of a portfolio solver: if multiple solvers finish at approximately the same time, which solution should we keep?

4.3 Threats to Validity

There are several threats to validity in our evaluation; we discuss the most important ones below. First, we considered a limited set of benchmarks and only the DFS strategy, so the results are not necessarily generalisable to other benchmarks and search strategies. Second, as discussed in Section 4, `KLEE` is highly nondeterministic; while we put a lot of effort into eliminating nondeterminism, there are still variations across runs, which may affect measurements. Third, our results may be influenced by specific implementation details in `KLEE`, so we cannot claim that they are generalisable to other symbolic execution engines. Finally, we used the default configuration of the SMT solvers made available through the `metaSMT` API. Different configuration options may change the relative performance of the solvers.

5 Portfolio Solving in Symbolic Execution

Symbolic execution can be enhanced with a parallel portfolio solver at different levels of granularity. The coarsest-grained option is to run multiple variants of the symbolic execution engine (in our case `KLEE`), each equipped with a different solver. This is essentially what our experiments in Sections 4.1 and 4.2 show. This has the advantage of having essentially no overhead (the runs are fully independent, and could even be run on separate machines) and being easy to deploy. Given a fixed time budget, one can run in parallel variants of `KLEE` configured with different solvers and select at the end the run that optimises a certain metric (e.g. number of executed instructions); or given a certain objective (e.g. a desired level of coverage), one could run concurrently the different variants of `KLEE` and abort execution when the first variant achieves that objective. The key advantage here is that *without any a priori knowledge of which solver is better on which benchmarks*, the user would obtain the results associated with the best-performing variant.

Our experiments in Sections 4.1 and 4.2 already show that this can be effective in practice: for instance, for the runs with caching presented in Figure 5, where the objective is to execute a given number of instructions, the user would obtain results as soon as `KLEE` with `STP` finishes for, say, `[` and `chmod` (where `STP` wins), and as soon as `KLEE` with `Z3` finishes for `echo` and `factor` (where `Z3` wins).

Another option for integrating portfolio solving in symbolic execution is at a finer level of granularity, e.g. at the level of individual solver queries, or of groups of consecutive queries. We are currently designing a portfolio solver at the query level, and while it is too early to report any results, we include below a discussion of the most important aspects that we have encountered.

As one moves to finer granularity, one creates opportunities for the portfolio-based variant of `KLEE` to behave better than *all* single-solver variants. At the coarse granularity of `KLEE` variants, one cannot perform better than the best variant; instead, using a portfolio solver at the query level, `KLEE` may perform significantly better than when equipped with the best individual solver, since different solvers may perform better on different queries.

On the other hand, at the query level, the performance overhead can be substantial, potentially negating in some cases the benefit of using a portfolio solver. This is particularly true in light of the fact that the vast majority of queries take very little time to complete, as discussed in Section 4.1. For such queries, the time spent spawning new threads or processes and monitoring their execution may be higher than the actual time spent in the SMT solvers. As a result, one idea is to start by running a single solver, and only if that solver does not return within a small time span (e.g. 0.1s), spawn the remaining solvers in the portfolio.

Related to the point above, we noticed that spawning threads versus spawning processes to run a solver can have a significant effect on performance. On the one hand, threads are more lightweight, and therefore incur less overhead: on several runs of `KLEE` configured with a single version of `STP` we observed significant speedups (varying from 1.25x to 1.93x) by simply switching from using a process to using a thread to run `STP`. On the other hand, using processes has the advantage that memory management is not an issue: on process exit, all the memory allocated by the solver is automatically freed by the operating system. This does not happen when threads are used, and we have observed that in many cases `KLEE` equipped with a portfolio of threaded solvers ends up consuming all available memory and starts thrashing.

Another important consideration is caching. As we discussed in Section 4.2, the actual counterexample values returned by a solver can have a significant influence on performance, so deciding what values to keep can be important. One option is to store the values of the first solver that returns with an answer. However, for queries where multiple solvers perform similarly, one might want to wait for a small additional amount of time to see if other solvers terminate too. If this happens, one can consider keeping all the counterexamples returned by different solvers, or selecting some of them: of course, keeping more counterexamples may increase the hit rate, but degrade performance. Some SMT solvers (and the SAT solvers they use) are incremental: to benefit from this aspect, it might also be important to wait for a short amount of time to allow more solvers to finish, as discussed above.

Finally, we consider the makeup of a portfolio solver. While including different SMT solvers is an obvious choice, based on our experience, we believe it is also important to consider different variants and versions of the same solver. While solvers may overall evolve for the better, given the nature of the problem, it is not uncommon to find queries on which newer versions of a solver perform worse. As a result, multiple versions of the same solver can be good candidates for a portfolio solver.

In addition, most solvers have a plethora of configuration options, which can have a significant impact on solving time. Selecting the right configuration parameters is a difficult decision, as it is often impossible to tell in advance which parameter values will perform better on which queries. Also, many modern SMT solvers are build on top of SAT solvers. Configuring a given SMT solver with different options and SAT solvers can provide additional candidates to include in the portfolio. Finally, SAT solvers have their own configuration options, which can be varied to create additional candidates.

6 Related Work

Constraint solving plays an important role in symbolic execution, and a significant effort has been invested in understanding and optimising constraint solving and constraint caching in a symbolic execution context, e.g. [2, 3, 19, 22, 25]. This paper provides additional information about the constraints encountered during symbolic execution (and in `KLEE` in particular), the effect of caching on solving time, and the relative performance of different solvers on several real benchmarks.

Portfolio solving has been explored in the past in the context of SAT solving [13, 24], SMT solving [23], and bounded model checking [9], among others. As far as we know, this is the first paper that reports on how different SMT solvers compare and could be combined in a portfolio solver in the context of symbolic execution.

Portfolio solving is a form of variant-based parallelization, which has been effectively used in the past to improve application performance, e.g. [8, 20, 21]. For instance, [20] proposes a general framework for competitive execution that targets multicore and multiprocessor systems, in which sequential applications are optimised by introducing competitive variants for parts of the program.

7 Conclusion

In this paper, we have discussed some of the most important characteristics of the constraints generated in symbolic execution, and identified several aspects that we believe are important for designing better SMT solvers for symbolic execution, and for combining multiple solvers using a portfolio-based approach. In particular, we have shown that counterexample values and caching can in some cases significantly affect constraint solving, and discussed several options for designing a portfolio solver for symbolic execution.

The reader can find additional information about our `KLEE` extension and experiments at `http://srg.doc.ic.ac.uk/projects/klee-multisolver`.

Acknowledgements. We would like to thank the program committee chairs of CAV 2013, Natasha Sharygina and Helmut Veith, for the opportunity to publish this invited paper. We are grateful to the `metaSMT` developers, in particular Heinz Riener and Finn Haedicke for their help with `metaSMT`. We would also like to thank Armin Biere for his help with `Boolector`. Finally, we thank Alastair Donaldson, Dan Liew, and Paul Marinescu, for their careful proofreading of our paper. This research is supported by the EPSRC grant EP/J00636X/1.

References

1. Brummayer, R., Biere, A.: Boolector: An efficient SMT solver for bit-vectors and arrays. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 174–177. Springer, Heidelberg (2009)
2. Cadar, C., Dunbar, D., Engler, D.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI 2008 (2008)

3. Cadar, C., Ganesh, V., Pawlowski, P., Dill, D., Engler, D.: EXE: Automatically generating inputs of death. In: CCS 2006 (2006)
4. Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C., Sen, K., Tillmann, N., Visser, W.: Symbolic execution for software testing in practice—preliminary assessment. In: ICSE Impact 2011 (2011)
5. Cadar, C., Pietzuch, P., Wolf, A.L.: Multiplicity computing: A vision of software engineering for next-generation computing platform applications. In: FoSER 2010 (2010)
6. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
8. Cledat, R., Kumar, T., Sreeram, J., Pande, S.: Opportunistic computing: A new paradigm for scalable realism on many-cores. In: HotPar 2009 (2009)
9. Cordeiro, L., Fischer, B., Marques-Silva, J.: SMT-based bounded model checking for embedded ANSI-C software. In: ASE 2009 (2009)
10. De Moura, L., Bjørner, N.: Satisfiability modulo theories: introduction and applications. Commun. ACM 54(9), 69–77 (2011)
11. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
12. Haedicke, F., Frehse, S., Fey, G., Große, D., Drechsler, R.: metaSMT: Focus on your application not on solver integration. In: DIFTS 2012 (2012)
13. Hamadi, Y., Sais, L.: ManySAT: a parallel SAT solver. JSAT 6, 245–262 (2009)
14. King, J.C.: Symbolic execution and program testing. CACM 19(7), 385–394 (1976)
15. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO 2004 (2004)
16. Marinescu, P.D., Cadar, C.: make test-zesti: A symbolic execution solution for improving regression testing. In: ICSE 2012 (2012)
17. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
18. Ranise, S., Tinelli, C.: The SMT-LIB format: An initial proposal. In: PDPAR 2003 (2003)
19. Sen, K., Marinov, D., Agha, G.: CUTE: A concolic unit testing engine for C. In: ESEC/FSE 2005 (2005)
20. Trachsel, O., Gross, T.R.: Variant-based competitive parallel execution of sequential programs. In: CF 2010 (2010)
21. Vajda, A., Stenstrom, P.: Semantic information based speculative parallel execution. In: PESPMA 2010 (2010)
22. Visser, W., Geldenhuys, J., Dwyer, M.B.: Green: reducing, reusing and recycling constraints in program analysis. In: FSE 2012 (2012)
23. Wintersteiger, C.M., Hamadi, Y., de Moura, L.: A concurrent portfolio approach to SMT solving. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 715–720. Springer, Heidelberg (2009)
24. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: SATzilla: portfolio-based algorithm selection for SAT. JAIR 32(1), 565–606 (2008)
25. Yang, G., Păsăreanu, C.S., Khurshid, S.: Memoized symbolic execution. In: ISSTA 2012 (2012)

Under-Approximating Cut Sets for Reachability in Large Scale Automata Networks

Loïc Paulevé[1], Geoffroy Andrieux[2], and Heinz Koeppl[1,3]

[1] ETH Zürich, Switzerland
[2] IRISA Rennes, France
[3] IBM Research - Zurich, Rueschlikon, Switzerland

Abstract. In the scope of discrete finite-state models of interacting components, we present a novel algorithm for identifying sets of local states of components whose activity is necessary for the reachability of a given local state. If all the local states from such a set are disabled in the model, the concerned reachability is impossible.

Those sets are referred to as cut sets and are computed from a particular abstract causality structure, so-called Graph of Local Causality, inspired from previous work and generalised here to finite automata networks. The extracted sets of local states form an under-approximation of the complete minimal cut sets of the dynamics: there may exist smaller or additional cut sets for the given reachability.

Applied to qualitative models of biological systems, such cut sets provide potential therapeutic targets that are proven to prevent molecules of interest to become active, up to the correctness of the model. Our new method makes tractable the formal analysis of very large scale networks, as illustrated by the computation of cut sets within a Boolean model of biological pathways interactions gathering more than 9000 components.

1 Introduction

With the aim of understanding and, ultimately, controlling physical systems, one generally constructs dynamical models of the known interactions between the components of the system. Because parts of those physical processes are ignored or still unknown, dynamics of such models aim at over-approximating the real system dynamics: any (observed) behaviour of the real system has to have a matching behaviour in the abstract model, the converse being potentially false. In such a setting, a valuable contribution of formal methods on abstract models of physical systems resides in the ability to prove the impossibility of particular behaviours.

Given a discrete finite-state model of interacting components, such as an automata network, we address here the computation of sets of local states of components that are necessary for reaching a local state of interest from a partially determined initial global state. Those sets are referred to as *cut sets*. Informally, each path leading to the reachability of interest has to involve, at one point, at least one local state of a cut set. Hence, disabling in the model all the local states referenced in one cut set should prevent the occurrence of the concerned reachability from delimited initial states. This is illustrated by Fig. 1.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 69–84, 2013.

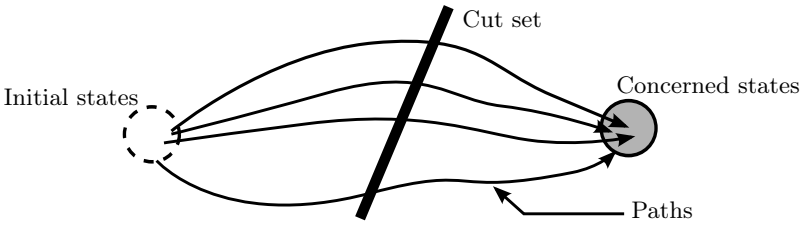


Fig. 1. A cut set is composed of local states that are involved in *all* paths from delimited initial states to concerned states. Disabling all the local states from such a cut set necessarily breaks the concerned reachability in the model.

Applied to a model of a biological system where the reachability of interest is known to occur, such cut sets provide potential coupled therapeutic targets to control the activity of a particular molecule (for instance using gene knock-in/out). The contrary implies that the abstract model is not an over-approximation of the concrete system.

Contribution. In this paper, we present a new algorithm to extract sets of local states that are necessary to achieve the concerned reachability within a finite automata network. Those sets are referred to as cut sets, and we limit ourselves to N-sets, *i.e.* having a maximum cardinality of N.

The finite automata networks we are considering are closely related to 1-safe Petri nets [1] having mutually exclusive places. They subsume Boolean and discrete networks [9,24,17,2], synchronous or asynchronous, that are widely used for the qualitative modelling of biological interaction networks.

A naive, but complete, algorithm could enumerate all potential candidate N-sets, disable each of them in the model, and then perform model-checking to verify if the targeted reachability is still verified. If not, the candidate N-set is a cut set. This would roughly leads to m^N tests, where m is the total number of local states in the automata network. Considering that the model-checking within automata networks is PSPACE-complete [5], this makes such an approach intractable on large networks.

The proposed algorithm aims at being tractable on systems composed of a very large number of interacting components, but each of them having a small number of local states. Our method principally overcomes two challenges: prevent a complete enumeration of candidate N-sets; and prevent the use of model-checking to verify if disabling a set of local states break the concerned reachability. It inherently handles partially-determined initial states: the resulting cut N-set of local states are proven to be necessary for the reachability of the local state of interest from *any* of the supplied global initial states.

The computation of the cut N-sets takes advantage of an abstraction of the formal model which highlights some steps that are necessary to occur prior to the verification of a given reachability property. This results in a causality structure called a *Graph of Local Causality* (GLC), which is inspired by [16], and that we generalise here to automata networks. Such a GLC has a size polynomial with

the total number of local states in the automata network, and exponential with the number of local states within one automata. Given a GLC, our algorithm propagates and combines the cut N-sets of the local states referenced in this graph by computing unions or products, depending on the disjunctive or conjunctive relations between the necessary conditions for their reachability. The algorithm is proven to converge in the presence of dependence cycles.

In order to demonstrate the scalability of our approach, we have computed cut N-sets within a very large Boolean model of a biological network relating more than 9000 components. Despite the highly combinatorial dynamics, a prototype implementation manages to compute up to the cut 5-sets within a few minutes. To our knowledge, this is the first time such a formal dynamical analysis has been performed on such a large dynamical model of biological system.

Related work and limitations. Cut sets are commonly defined upon graphs as set of edges or vertices which, if removed, disconnect a given pair of nodes [21]. For our purpose, this approach could be directly applied to the global transition graph to identify local states or transitions for which the removal would disconnect initial states from the targeted states. However, the combinatorial explosion of the state space would make it intractable for large interacting systems.

The aim of the presented method is somehow similar to the generation of minimal cut sets in fault trees [13,22] used for reliability analysis, as the structure representing reachability causality contains both *and* and *or* connectors. However, the major difference is that we are here dealing with cyclic directed graphs which prevents the above mentioned methods to be straightforwardly applied.

Klamt *et al.* have developed a complete method for identifying minimal cut sets (also called intervention sets) dedicated to biochemical reactions networks, hence involving cycling dependencies [10]. This method has been later generalised to Boolean models of signalling networks [19]. Those algorithms are mainly based on the enumeration of possible candidates, with techniques to reduce the search space, for instance by exploiting symmetry of dynamics. Whereas intervention sets of [10,19] can contain either local states or reactions, our cut sets are only composed of local states.

Our method follows a different approach than [10,19] by not relying on candidate enumeration but computing the cut sets directly on an abstract structure derived statically from the model, which should make tractable the analysis of very large networks. The comparison with [19] is detailed in Subsect. 4.1.

In addition, our method is generic to any automata network, but relies on an abstract interpretation of dynamics which leads to under-approximating the cut sets for reachability: by ignoring certain dynamical constraints, the analysis can miss several cut sets and output cut sets that are not minimal for the concrete model. Finally, although we focus on finding the cut sets for the reachability of only *one* local state, our algorithm computes the cut sets for the (independent) reachability of all local states referenced in the GLC.

Outline. Sect. 2 introduces a generic characterisation of the *Graph of Local Causality* with respect to automata networks; Sect. 3 states and sketches the proof of the algorithm for extracting a subset of N-sets of local states necessary for the reachability of a given local state. Sect. 4 discusses the application to systems biology by comparing with the related work and applying our new method to a very large scale model of biological interactions. Finally, Sect. 5 discusses the results presented and some of their possible extensions.

Notations. $\wedge$ and $\vee$ are the usual logical *and* and *or* connectors. $[1;n] = \{1,\cdots,n\}$. Given a finite set A, $\#A$ is the cardinality of A; $\wp(A)$ is the power set of A; $\wp^{\leq N}(A)$ is the set of all subsets of A with cardinality at most N. Given sets $A^1,\ldots,A^n$, $\bigcup_{i\in[1;n]} A^i$ is the union of those sets, with the empty union $\bigcup_\emptyset \triangleq \emptyset$; and $A^1 \times \cdots \times A^n$ is the usual Cartesian product. Given sets of sets $B^1,\ldots,B^n \in \wp(\wp(A))$, $\widetilde{\prod}_{i\in[1;n]} B^i \triangleq B^1 \tilde{\times} \cdots \tilde{\times} B^n \in \wp(\wp(A))$ is the *sets of sets product* where $\{e_1,\ldots,e_n\}\tilde{\times}\{e'_1,\ldots,e'_m\} \triangleq \{e_i \cup e'_j \mid i \in [1;n] \wedge j \in [1;m]\}$. In particular $\forall (i,j) \in [1;n]\times[1;m]$, $B^i \tilde{\times} B^j = B^j \tilde{\times} B^i$ and $\emptyset \tilde{\times} B^i = \emptyset$. The empty sets of sets product $\widetilde{\prod}_\emptyset \triangleq \{\emptyset\}$. If $M : A \mapsto B$ is a mapping from elements in A to elements in B, $M(a)$ is the value in B mapped to $a \in A$; $M\{a \mapsto b\}$ is the mapping M where $a \in A$ now maps to $b \in B$.

2 Graph of Local Causality

We first give basic definitions of automata networks, local state disabling, context and local state reachability; then we define the local causality of an objective (local reachability), and the *Graph of Local Causality*. A simple example is given at the end of the section.

2.1 Finite Automata Networks

We consider a network of automata $(\Sigma, S, \mathcal{L}, T)$ which relates a finite number of interacting finite state automata Σ (Def. 1). The global state of the system is the gathering of the local state of composing automata. A transition can occur if and only if all the local states sharing a common transition label $\ell \in L$ are present in the global state $s \in S$ of the system. Such networks characterize a class of 1-safe Petri Nets [1] having groups of mutually exclusive places, acting as the automata. They allow the modelling of Boolean networks and their discrete generalisation, having either synchronous or asynchronous transitions.

Definition 1 (Automata Network $(\Sigma, S, \mathcal{L}, T)$). *An automata network is defined by a tuple $(\Sigma, S, \mathcal{L}, T)$ where*

- *$\Sigma = \{a, b, \ldots, z\}$ is the finite set of automata identifiers;*
- *For any $a \in \Sigma$, $S(a) = [1; k_a]$ is the finite set of local states of automaton a; $S = \prod_{a\in\Sigma}[1;k_a]$ is the finite set of global states.*
- *$\mathcal{L} = \{\ell_1,\ldots,\ell_m\}$ is the finite set of transition labels;*

– $T = \{a \mapsto T_a \mid a \in \Sigma\}$, *where* $\forall a \in \Sigma, T_a \subset [1; k_a] \times \mathcal{L} \times [1; k_a]$, *is the mapping from automata to their finite set of local transitions.*

We note $i \xrightarrow{\ell} j \in T(a) \overset{\Delta}{\Leftrightarrow} (i, \ell, j) \in T_a$ *and* $a_i \xrightarrow{\ell} a_j \in T \overset{\Delta}{\Leftrightarrow} i \xrightarrow{\ell} j \in T(a)$.
$\forall \ell \in \mathcal{L}$, *we note* ${}^\bullet\ell \overset{\Delta}{=} \{a_i \mid a_i \xrightarrow{\ell} a_j \in T(a)\}$ *and* $\ell^\bullet \overset{\Delta}{=} \{a_j \mid a_i \xrightarrow{\ell} a_j \in T(a)\}$.
The set of local states is defined as $\mathbf{LS} \overset{\Delta}{=} \{a_i \mid a \in \Sigma \wedge i \in [1; k_a]\}$.
The global transition relation $\rightarrow \subset S \times S$ *is defined as:*

$$s \rightarrow s' \overset{\Delta}{\Leftrightarrow} \exists \ell \in \mathcal{L} : \forall a_i \in {}^\bullet\ell, s(a) = a_i \wedge \forall a_j \in \ell^\bullet, s'(a) = a_j \\ \wedge \forall b \in \Sigma, S(b) \cap {}^\bullet\ell = \emptyset \Rightarrow s(b) = s'(b).$$

Given an automata network $\mathcal{S}ys = (\Sigma, S, \mathcal{L}, T)$ and a subset of its local states $ls \subseteq \mathbf{LS}$, $\mathcal{S}ys \ominus ls$ refers to the system where all the local states ls have been disabled, i.e. they can not be involved in any transition (Def. 2).

Definition 2 (Local states disabling). *Given* $\mathcal{S}ys = (\Sigma, S, \mathcal{L}, T)$ *and* $ls \in \wp(\mathbf{LS})$, $\mathcal{S}ys \ominus ls \overset{\Delta}{=} (\Sigma, S, \mathcal{L}', T')$ *where* $\mathcal{L}' = \{\ell \in \mathcal{L} \mid ls \cap {}^\bullet\ell = \emptyset\}$ *and* $T' = \{a_i \xrightarrow{\ell} a_j \in T \mid \ell \in \mathcal{L}'\}$.

From a set of acceptable initial states delimited by a *context* ς (Def. 3), we say a given local state $a_j \in \mathbf{LS}$ is reachable if and only if there exists a finite number of transitions in $\mathcal{S}ys$ leading to a global state where a_j is present (Def. 4).

Definition 3 (Context ς). *Given a network* $(\Sigma, S, \mathcal{L}, T)$, *a context* ς *is a mapping from each automaton* $a \in \Sigma$ *to a non-empty subset of its local states:* $\forall a \in \Sigma, \varsigma(a) \in \wp(S(a)) \wedge \varsigma(a) \neq \emptyset$.

Definition 4 (Local state reachability). *Given a network* $(\Sigma, S, \mathcal{L}, T)$ *and a context* ς, *the local state* $a_j \in \mathbf{LS}$ *is* reachable *from* ς *if and only if* $\exists s_0, \ldots, s_m \in S$ *such that* $\forall a \in \Sigma, s_0(a) \in \varsigma(a)$, *and* $s_0 \rightarrow \cdots \rightarrow s_m$, *and* $s_m(a) = j$.

2.2 Local Causality

Locally reasoning within one automaton a, the global reachability of a_j from ς can be expressed as the reachability of a_j from a local state $a_i \in \varsigma(a)$. This local reachability specification is referred to as an *objective* noted $a_i \rightarrow^* a_j$ (Def. 5).

Definition 5 (Objective). *Given a network* $(\Sigma, S, \mathcal{L}, T)$, *the reachability of local state* a_j *from* a_i *is called an* objective *and is denoted* $a_i \rightarrow^* a_j$. *The set of all objectives is referred to as* $\mathbf{Obj} \overset{\Delta}{=} \{a_i \rightarrow^* a_j \mid (a_i, a_j) \in \mathbf{LS} \times \mathbf{LS}\}$.

Given an objective $P = a_i \rightarrow^* a_j \in \mathbf{Obj}$, we define $\mathsf{sol}(P)$ the *local causality* of P (Def. 6): each $ls \in \mathsf{sol}(P)$ is a set of local states that may be involved for the reachability of a_j from a_i; ls is referred to as a (local) solution for P. $\mathsf{sol}(P)$ is sound as soon as the disabling of at least one local state in *each* solution makes the reachability of a_j impossible from any global state containing a_i (Property 1). It implies that if $\mathsf{sol}(P) = \{\{a_i\} \cup ls^1, \ldots, ls^m\}$ is sound, $\mathsf{sol}'(P) = \{ls^1, \ldots, ls^m\}$ is also sound. $\mathsf{sol}(a_i \rightarrow^* a_j) = \emptyset$ implies that a_j can never be reached from a_i, and $\forall a_i \in \mathbf{LS}, \mathsf{sol}(a_i \rightarrow^* a_i) \overset{\Delta}{=} \{\emptyset\}$.

Definition 6. $\mathsf{sol} : \mathbf{Obj} \mapsto \wp(\wp(\mathbf{LS}))$ *is a mapping from objectives to sets of sets of local states such that* $\forall P \in \mathbf{Obj}, \forall ls \in \mathsf{sol}(P), \nexists ls' \in \mathsf{sol}(P), ls \neq ls'$ *such that* $ls' \subset ls$*. The set of these mappings is noted* $\mathbf{Sol} \triangleq \{\langle P, ls\rangle \mid ls \in \mathsf{sol}(P)\}$.

Property 1 (sol *soundness).* $\mathsf{sol}(a_i \to^* a_j) = \{ls^1, \ldots, ls^n\}$ is a sound set of solutions for the network $\mathcal{S}ys = (\Sigma, S, \mathcal{L}, T)$ if and only if $\forall kls \in \widetilde{\prod}_{i\in[1;n]}\{ls^i\}$, a_j is not reachable in $\mathcal{S}ys \ominus kls$ from any state $s \in S$ such that $s(a) = i$.

In the rest of this paper we assume that Property 1 is satisfied, and consider sol computation out of the scope of this paper.

Nevertheless, we briefly describe a construction of a sound $\mathsf{sol}(a_i \to^* a_j)$ for an automata network $(\Sigma, S, \mathcal{L}, T)$; an example is given at the end of this section. This construction generalises the computation of GLC from the Process Hitting framework, a restriction of network of automata depicted in [16]. For each acyclic sequence $a_i \xrightarrow{\ell_1} \ldots \xrightarrow{\ell_m} a_j$ of local transitions in $T(a)$, and by defining $\mathsf{ext}_a(\ell) \triangleq \{b_j \in \mathbf{LS} \mid b_j \xrightarrow{\ell} b_k \in T, b \neq a\}$, we set $ls \in \widetilde{\prod}_{\ell\in\{\ell_1,\ldots,\ell_m \mid \mathsf{ext}_a(\ell)\neq\emptyset\}}\{\mathsf{ext}_a(\ell)\} \Rightarrow ls \in \mathsf{sol}(a_i \to^* a_j)$, up to supersets removing. One can easily show that Property 1 is verified with such a construction. The complexity of this construction is exponential in the number of local states within one automaton and polynomial in the number of automata. Alternative constructions may also provide sound (and not necessarily equal) sol.

2.3 Graph of Local Causality

Given a local state $a_j \in \mathbf{LS}$ and an initial context ς, the reachability of a_i is equivalent to the realization of any objective $a_i \to^* a_j$, with $a_i \in \varsigma(a)$. By definition, if a_j is reachable from ς, there exists $ls \in \mathsf{sol}(a_i \to^* a_j)$ such that, $\forall b_k \in ls$, b_k is reachable from ς.

The (directed) *Graph of Local Causality* (GLC, Def. 7) relates this recursive reasoning from a given set of local states $\omega \subseteq \mathbf{LS}$ by linking every local state a_j to all objectives $a_i \to^* a_j, a_i \in \varsigma(a)$; every objective P to its solutions $\langle P, ls\rangle \in \mathbf{Sol}$; every solution $\langle P, ls\rangle$ to its local states $b_k \in ls$. A GLC is said to be sound if sol is sound for all referenced objectives (Property 2).

Definition 7 (Graph of Local Causality). *Given a context* ς *and a set of local states* $\omega \subseteq \mathbf{LS}$*, the* Graph of Local Causality *(GLC)* $\mathcal{A}^\omega_\varsigma \triangleq (V^\omega_\varsigma, E^\omega_\varsigma)$*, with* $V^\omega_\varsigma \subseteq \mathbf{LS} \cup \mathbf{Obj} \cup \mathbf{Sol}$ *and* $E^\omega_\varsigma \subseteq V^\omega_\varsigma \times V^\omega_\varsigma$*, is the smallest structure satisfying:*

$$\begin{aligned}
\omega &\subseteq V^\omega_\varsigma \\
a_i \in V^\omega_\varsigma \cap \mathbf{LS} &\Leftrightarrow \{(a_i, a_j \to^* a_i) \mid a_j \in \varsigma\} \subseteq E^\omega_\varsigma \\
a_i \to^* a_j \in V^\omega_\varsigma \cap \mathbf{Obj} &\Leftrightarrow \{(a_i \to^* a_j, \langle a_i \to^* a_j, ls\rangle) \mid \langle a_i \to^* a_j, ls\rangle \in \mathbf{Sol}\} \subseteq E^\omega_\varsigma \\
\langle P, ls\rangle \in V^\omega_\varsigma \cap \mathbf{Sol} &\Leftrightarrow \{(\langle P, ls\rangle, a_i) \mid a_i \in ls\} \subseteq E^\omega_\varsigma \quad .
\end{aligned}$$

Property 2 (Sound Graph of Local Causality). A GLC $\mathcal{A}^\omega_\varsigma$ is *sound* if, $\forall P \in V^\omega_\varsigma \cap \mathbf{Obj}$, $\mathsf{sol}(P)$ is sound.

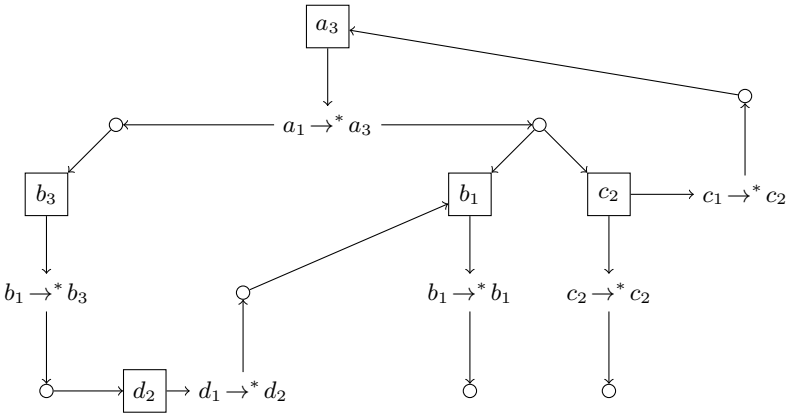


Fig. 2. Example of Graph of Local Causality that is sound for the automata network defined in Example 1

This structure can be constructed starting from local states in ω and by iteratively adding the imposed children. It is worth noticing that this graph can contain cycles. In the worst case, $\#V_\varsigma^\omega = \#\mathbf{LS} + \#\mathbf{Obj} + \#\mathbf{Sol}$ and $\#E_\varsigma^\omega = \#\mathbf{Obj} + \#\mathbf{Sol} + \sum_{\langle P, ls\rangle \in \mathbf{Sol}} \#ls$.

Example 1. Fig. 2 shows an example of GLC. Local states are represented by boxed nodes and elements of **Sol** by small circles.

For instance, such a GLC is sound for the following automata network $(\Sigma, S, \mathcal{L}, T)$, with initial context $\varsigma = \{a \mapsto \{1\}; b \mapsto \{1\}; c \mapsto \{1,2\}; d \mapsto \{2\}\}$:

$$\Sigma = \{a, b, c, d\} \qquad \mathcal{L} = \{\ell_1, \ell_2, \ell_3, \ell_4, \ell_5, \ell_6\}$$
$$S(a) = [1;3] \qquad T(a) = \{1 \xrightarrow{\ell_2} 2; 2 \xrightarrow{\ell_3} 3; 1 \xrightarrow{\ell_1} 3; 3 \xrightarrow{\ell_4} 2\}$$
$$S(b) = [1;3] \qquad T(b) = \{1 \xrightarrow{\ell_2} 2; 1 \xrightarrow{\ell_5} 3; 1 \xrightarrow{\ell_6} 1; 3 \xrightarrow{\ell_1} 2\}$$
$$S(c) = [1;2] \qquad T(c) = \{1 \xrightarrow{\ell_4} 2; 2 \xrightarrow{\ell_3} 1\}$$
$$S(d) = [1;2] \qquad T(d) = \{1 \xrightarrow{\ell_6} 2; 2 \xrightarrow{\ell_5} 1\}$$

For example, within automata a, there are two acyclic sequences from 1 to 3: $1 \xrightarrow{\ell_2} 2 \xrightarrow{\ell_3} 3$ and $1 \xrightarrow{\ell_1} 3$. Hence, if a_3 is reached from a_1, then necessarily, one of these two sequences has to be used (but not necessarily consecutively). For each of these transitions, the transition label is shared by exactly one local state in another automaton: b_1, c_2, b_3 for ℓ_2, ℓ_3, ℓ_1, respectively. Therefore, if a_3 is reached from a_1, then necessarily either both b_1 and c_2, or b_3 have been reached before. Hence $\mathsf{sol}(a_1 \to^* a_3) = \{\{b_1, c_2\}, \{b_3\}\}$ is sound, as disabling either b_1 and b_3, or c_2 and b_3, would remove any possibility to reach a_3 from a_1.

3 Necessary Local States for Reachability

We assume a global sound GLC $\mathcal{A}^\omega_\varsigma = (V^\omega_\varsigma, E^\omega_\varsigma)$, with the usual accessors for the direct relations of nodes:

$$\mathsf{children} : V^\omega_\varsigma \mapsto \wp(V^\omega_\varsigma) \qquad \mathsf{parents} : V^\omega_\varsigma \mapsto \wp(V^\omega_\varsigma)$$
$$\mathsf{children}(n) \triangleq \{m \in V^\omega_\varsigma \mid (n,m) \in E^\omega_\varsigma\} \quad \mathsf{parents}(n) \triangleq \{m \in V^\omega_\varsigma \mid (m,n) \in E^\omega_\varsigma\}$$

Given a set of local states $\mathcal{O}bs \subseteq \mathbf{LS}$, this section introduces an algorithm computing upon $\mathcal{A}^\omega_\varsigma$ the set $\mathbb{V}(a_i)$ of minimal cut N-sets of local states in $\mathcal{O}bs$ that are necessary for the independent reachability of each local state $a_i \in \mathbf{LS} \cap V^\omega_\varsigma$. The minimality criterion actually states that $\forall ls \in \mathbb{V}(a_i)$, there is no different $ls' \in \mathbb{V}(a_i)$ such that $ls' \subset ls$.

Assuming a first valuation $\mathbb{V}$ (Def. 8) associating to each node its cut N-sets, the cut N-sets for the node n can be refined using $\mathsf{update}(\mathbb{V}, n)$ (Def. 9):

- if n is a solution $\langle P, ls\rangle \in \mathbf{Sol}$, it is sufficient to prevent the reachability of *any* local state in ls to cut n; therefore, the cut N-sets results from the union of the cut N-sets of n children (all local states).
- If n is an objective $P \in \mathbf{Obj}$, all its solutions (in $\mathsf{sol}(P)$) have to be cut in order to ensure that P is not realizable: hence, the cut N-sets result from the product of children cut N-sets (all solutions).
- If n is a local state a_i, it is sufficient to cut all its children (all objectives) to prevent the reachability of a_i from any state in the context ς. In addition, if $a_i \in \mathcal{O}bs$, $\{a_i\}$ is added to the set of its cut N-sets.

Definition 8 (Valuation $\mathbb{V}$). *A* valuation $\mathbb{V} : V^\omega_\varsigma \mapsto \wp(\wp^{\leq N}(\mathcal{O}bs))$ *is a mapping from each node of* $\mathcal{A}^\omega_\varsigma$ *to a set of* N*-sets of local states.* $\mathbf{Val}$ *is the set of all valuations.* $\mathbb{V}_0 \in \mathbf{Val}$ *refers to the valuation such that* $\forall n \in V^\omega_\varsigma, \mathbb{V}_0(n) = \emptyset$.

Definition 9 ($\mathsf{update} : \mathbf{Val} \times V^\omega_\varsigma \mapsto \mathbf{Val}$).

$$\mathsf{update}(\mathbb{V}, n) \triangleq \begin{cases} \mathbb{V}\{n \mapsto \zeta^N(\bigcup_{m\in\mathsf{children}(n)} \mathbb{V}(m))\} & \text{if } n \in \mathbf{Sol} \\ \mathbb{V}\{n \mapsto \zeta^N(\widetilde{\prod}_{m\in\mathsf{children}(n)} \mathbb{V}(m))\} & \text{if } n \in \mathbf{Obj} \\ \mathbb{V}\{n \mapsto \zeta^N(\widetilde{\prod}_{m\in\mathsf{children}(n)} \mathbb{V}(m))\} & \text{if } n \in \mathbf{LS} \setminus \mathcal{O}bs \\ \mathbb{V}\{n \mapsto \zeta^N(\{\{a_i\}\} \cup \widetilde{\prod}_{m\in\mathsf{children}(n)} \mathbb{V}(m))\} & \text{if } n \in \mathbf{LS} \cap \mathcal{O}bs \end{cases}$$

where $\zeta^N(\{e_1, \ldots, e_n\}) \triangleq \{e_i \mid i \in [1;n] \wedge \#e_i \leq N \wedge \nexists j \in [1;n], j \neq i, e_j \subset e_i\}$, e_i *being sets,* $\forall i \in [1;n]$.

Starting with $\mathbb{V}_0$, one can repeatedly apply update on each node of $\mathcal{A}^\omega_\varsigma$ to refine its valuation. Only nodes where one of their children value has been modified should be considered for updating.

Hence, the order of nodes updates should follow the topological order of the GLC, where children have a lower rank than their parents (i.e., children are treated before their parents). If the graph is actually acyclic, then it is sufficient

Algorithm 1. $\mathcal{A}^{\omega}_{\varsigma}$-MINIMAL-CUT-NSETS

```
1: M ← V_ς^ω
2: 𝕍 ← 𝕍_0
3: while M ≠ ∅ do
4:     n ← arg min_{m∈M}{rank(m)}
5:     M ← M \ {n}
6:     𝕍' ← update(𝕍, n)
7:     if 𝕍'(n) ≠ 𝕍(n) then
8:         M ← M ∪ parents(n)
9:     end if
10:    𝕍 ← 𝕍'
11: end while
12: return 𝕍
```

to update the value of each node only once. In the general case, *i.e.* in the presence of Strongly Connected Components (SCCs) — nodes belonging to the same SCC have the same rank —, the nodes within a SCC have to be iteratively updated until the convergence of their valuation.

Algorithm 1 formalizes this procedure where $\mathsf{rank}(n)$ refers to the topological rank of n, as it can be derived from Tarjan's strongly connected components algorithm [23], for example. The node $n \in V^{\omega}_{\varsigma}$ to be updated is selected as being the one having the least rank amongst the nodes to update (delimited by $\mathcal{M}$). In the case where several nodes with the same lowest rank are in $\mathcal{M}$, they can be either arbitrarily or randomly picked. Once picked, the value of n is updated. If the new valuation of n is different from the previous, the parents of n are added to the list of nodes to update (lines 6-8 in Algorithm 1).

Lemma 1 states the convergence of Algorithm 1 and Theorem 1 its correctness: for each local state $a_i \in V^{\omega}_{\varsigma} \cap \mathbf{LS}$, each set of local states $kls \in \mathbb{V}(a_i)$ (except $\{a_i\}$ singleton) references the local states that are all necessary to reach prior to the reachability of a_i from any state in ς. Hence, if all the local states in kls are disabled in $\mathcal{S}ys$, a_i is not reachable from any state in ς.

Lemma 1. $\mathcal{A}^{\omega}_{\varsigma}$-MINIMAL-CUT-NSETS *always terminates.*

Proof. Remarking that $\wp(\wp^{\leq N}(\mathcal{O}bs))$ is finite, defining a partial ordering such that $\forall v, v' \in \wp(\wp^{\leq N}(\mathcal{O}bs)), v \succeq v' \overset{A}{\Leftrightarrow} \zeta^N(v) = \zeta^N(v \cup v')$, and noting $\mathbb{V}^k \in \mathbf{Val}$ the valuation after k iterations of the algorithm, it is sufficient to prove that $\mathbb{V}^{k+1}(n) \succeq \mathbb{V}^k(n)$. Let us define $v_1, v_2, v'_1, v'_2 \in \wp(\wp^{\leq N}(\mathcal{O}bs))$ such that $v_1 \succeq v'_1$ and $v_2 \succeq v'_2$. We can easily check that $v_1 \cup v_2 \succeq v'_1 \cup v'_2$ (hence proving the case when $n \in \mathbf{Sol}$). As $\zeta^N(v_1) = \zeta^N(v_1 \cup v'_1) \Leftrightarrow \forall e'_1 \in v'_1, \exists e_1 \in v_1 : e_1 \subseteq e'_1$, we obtain that $\forall (e'_1, e'_2) \in v'_1 \times v'_2, \exists (e_1, e_2) \in v_1 \times v_2 : e_1 \subseteq e'_1 \wedge e_2 \subseteq e'_2$. Hence $e_1 \cup e_2 \subseteq e'_1 \cup e'_2$, therefore $\zeta^N(v_1 \tilde{\times} v_2 \cup v'_1 \tilde{\times} v'_2) = \zeta^N(v_1 \tilde{\times} v_2)$, i.e. $v_1 \tilde{\times} v_2 \succeq v'_1 \tilde{\times} v'_2$; which proves the cases when $n \in \mathbf{Obj} \cup \mathbf{LS}$.

Theorem 1. *Given a GLC* $\mathcal{A}^{\omega}_{\varsigma} = (V^{\omega}_{\varsigma}, E^{\omega}_{\varsigma})$ *which is sound for the automata network* $\mathcal{S}ys$*, the valuation* $\mathbb{V}$ *computed by* $\mathcal{A}^{\omega}_{\varsigma}$-MINIMAL-CUT-NSETS *verifies:* $\forall a_i \in \mathbf{LS} \cap V^{\omega}_{\varsigma}, \forall kls \in \mathbb{V}(a_i) \setminus \{\{a_i\}\}$, a_j *is not reachable from* ς *within* $\mathcal{S}ys \ominus kls$.

Table 1. Result of the execution of Algorithm 1 on the GLC in Fig. 2

Node	rank	$\mathbb{V}$
$\langle b_1 \to^* b_1, \emptyset \rangle$	1	$\emptyset$
$b_1 \to^* b_1$	2	$\emptyset$
b_1	3	$\{\{b_1\}\}$
$\langle d_1 \to^* d_2, \{b_1\} \rangle$	4	$\{\{b_1\}\}$
$d_1 \to^* d_2$	5	$\{\{b_1\}\}$
d_2	6	$\{\{b_1\}, \{d_2\}\}$
$\langle b_1 \to^* b_3, \{d_2\} \rangle$	7	$\{\{b_1\}, \{d_2\}\}$
$b_1 \to^* b_3$	8	$\{\{b_1\}, \{d_2\}\}$
b_3	9	$\{\{b_1\}, \{b_3\}, \{d_2\}\}$
$\langle a_1 \to^* a_3, \{b_3\} \rangle$	10	$\{\{b_1\}, \{b_3\}, \{d_2\}\}$
$\langle c_2 \to^* c_2, \emptyset \rangle$	11	$\emptyset$
$c_2 \to^* c_2$	12	$\emptyset$
c_2	13	$\{\{c_2\}\}$
$\langle a_1 \to^* a_3, \{b_1, c_2\} \rangle$	13	$\{\{b_1\}, \{c_2\}\}$
$a_1 \to^* a_3$	13	$\{\{b_1\}, \{b_3, c_2\}, \{c_2, d_2\}\}$
a_3	13	$\{\{a_3\}, \{b_1\}, \{b_3, c_2\}, \{c_2, d_2\}\}$
$\langle c_1 \to^* c_2, \{a_3\} \rangle$	13	$\{\{a_3\}, \{b_1\}, \{b_3, c_2\}, \{c_2, d_2\}\}$

Proof. By recurrence on the valuations $\mathbb{V}$: the above property is true at each iteration of the algorithm.

Example 2. Table 1 details the result of the execution of Algorithm 1 on the GLC defined in Fig. 2. Nodes receive a topological rank, identical ranks implying the belonging to the same SCC. The (arbitrary) scheduling of the updates of nodes within a SCC follows the order in the table. In this particular case, nodes are all visited once, as $\mathbb{V}(\langle c_2 \to^* c_2, \emptyset \rangle) \tilde{\times} \mathbb{V}(\langle c_1 \to^* c_2, \{a_3\} \rangle) = \emptyset$ (hence $\mathsf{update}(\mathbb{V}, c_2)$ does not change the valuation of c_2). Note that in general, several iterations of update may be required to reach a fixed point.

It is worth noticing that the GLC abstracts several dynamical constraints in the underlying automata networks, such as the ordering of transitions, or the synchronous updates of the global state. In that sense, GLC over-approximates the dynamics of the network, and the resulting cut sets are under-approximating the complete cut sets of the concrete model: any computed cut sets is a superset of a complete cut set (potentially equal).

4 Application to Systems Biology

Automata networks, as presented in Def. 1, subsume Boolean and discrete networks, synchronous and asynchronous, that are widely used for the qualitative modelling of dynamics of biological networks [9,24,17,2,7,18,6].

A cut set, as extracted by our algorithm, informs that at least one of the component in the cut set has to be present in the specified local state in order to achieve the wanted reachability. A local state can represent, for instance,

an active transcription factor or the absence of a certain protein. It provides potential therapeutic targets if the studied reachability is involved in a disease by preventing all the local states of a cut set to act, for instance using gene knock-out or knock-in techniques.

We first discuss and compare our methodology with the *intervention sets* analysis within biological models developed by S. Klamt *et al.*, and provide some benchmarks on a few examples.

Thanks to the use of the intermediate GLC and to the absence of candidate enumeration, our new method makes tractable the cut sets analysis on very large models. We present a recent application of our results to the analysis of a very large scale Boolean model of biological pathway interactions involving 9000 components. To our knowledge, this is the first attempt of a formal dynamical analysis on such a large scale model.

4.1 Related Work

The general related work having been discussed in Sect. 1, we deepen here the comparison of our method with the closest related work: the analysis of *Intervention Sets* (ISs) [19]. Cut sets and ISs have a reversed logic: an IS specifies local states to enforce in order to ensure a particular behaviour to occur; a cut set specifies local states to disable in order to prevent a particular behaviour to occur. In the scope of Boolean models of signalling networks, ISs are computed for the reachability of a given fixed point (steady state) which can be partially defined. Their method is complete: all minimal ISs are computed.

Nevertheless, the semantics and the computation of ISs have some key differences with our computed cut sets. First, they focus only on the reachability of (logical) steady states, which is a stronger condition than the transient reachability that we are considering. Then, the steady states are computed using a three-valued logic which allows to cope with undefined (initial) local states, but which is different from the notion of context that we use in this paper for specifying the initial condition.

Such differences make difficult a proper comparison of inferred cut sets. We can however expect that any cut sets found by our method has a corresponding IS in the scope of Boolean networks with a single initial state.

To give a practical insight on the relation between the two methods, we compare the results for two signalling networks, both between a model specified with CellNetAnalyser [11] to compute ISs and a model specified in the Process Hitting framework, a particular restriction of asynchronous automata networks [15], to compute our cut sets. Process Hitting models have been built in order to over-approximate the dynamics considered for the computation of ISs[1].

Tcell. Applied to a model of the T-cell receptor signalling between 40 components [12], we are interested in preventing the activation of the transcription factor $AP1$. For an instance of initial conditions, and limiting the computations to 3-sets, 31 ISs have been identified (28 1-sets, 3 2-sets, 0 3-set), whereas our

[1] Models and scripts available at `http://loicpauleve.name/cutsets.tbz2`

algorithm found 29 cut sets (21 1-sets and 8 2-sets), which are all matching an IS (23 are identical, 6 strictly including ISs). ISs are computed in 0.69s while our algorithm under-approximates the cut sets in 0.006s. Different initial states give comparable results.

Egfr. Applied to a model of the epidermal growth factor receptor signalling pathway of 104 components [18], we are interested in preventing the activation of the transcription factor $AP1$. For an instance of initial conditions, and limiting the computations to 3-sets, 25 ISs have been identified (19 1-sets, 3 2-sets, 3 3-sets), whereas our algorithm found 14 cut sets (14 1-sets), which are all included in the ISs. ISs are computed in 98s while our algorithm under-approximates the cut sets in 0.004s. Different initial states give comparable results.

As expected with the different semantics of models and cut sets, resulting ISs matches all the cut sets identified by our algorithm, and provides substantially more sets. The execution time is much higher for ISs as they rely on candidate enumeration in order to provide complete results, whereas our method was designed to prevent such an enumeration but under-approximates the cut sets.

In order to appreciate the under-approximation done by our method at a same level of abstraction and with identical semantics, we compare the cut sets identified by our algorithm with the cut sets obtained using a naive, but complete, computation. The naive computation enumerates all cut set candidates and, for each of them, disable the local states in the model and perform model-checking to verify if the target local state is still reachable. In the particular case of these two models, and limiting the cut sets to 3 and 2-sets respectively for the sake of tractability, no additional cut set has been uncovered by the complete method. Such a good under-approximation could be partially explained by the restrictions imposed on the causality by the Process Hitting framework, making the GLC a tight over-approximation of the dynamics [16].

4.2 Very Large Scale Application to Pathway Interactions

In order to support the scalability of our approach, we apply the proposed algorithm to a very large model of biological interactions, actually extracted from the PID database [20] referencing various influences (complex formation, inductions (activations) and inhibitions, transcriptional regulation, etc.) between more than 9000 biological components (proteins, genes, ions, etc.).

Amongst the numerous biological components, the activation of some of them are known to control key mechanisms of the cell dynamics. Those activations are the consequence of intertwining signalling pathways and depend on the environment of the cell (represented by the presence of certain *entry-point* molecules). Uncovering the environmental and intermediate components playing a major role in these signalling dynamics is of great biological interest.

The full PID database has been interpreted into the Process Hitting framework, a subclass of asynchronous automata networks, from which the derivation of the GLC has been addressed in previous work [16]. The obtained model gathers components representing either biological entities modelled as boolean value (absent or present), or logical complexes. When a biological component has several

competing regulators, the precise cooperations are not detailed in the database, so we use of two different interpretations: all (resp. one of) the activators and none (resp. all but one of) the inhibitors have to be present in order to make the target component present. This leads to two different discrete models of PID that we refer to as `whole_PID_AND` and `whole_PID_OR`, respectively.

Focusing on `whole_PID_OR`, the Process Hitting model relates more than 21000 components, either biological or logical, containing between 2 and 4 local states. Such a system could actually generate 2^{33874} states. 3136 components act as environment specification, which in our boolean interpretation leads to 2^{3136} possible initial states, assuming all other components start in the absent state.

We focus on the (independent) reachability of active SNAIL transcription factor, involved in the epithelial to mesenchymal transition [14], and of active p15INK4b and p21CIP1 cyclin-dependent kinase inhibitors involved in cell cycle regulation [3]. The GLC relates 20045 nodes, including 5671 component local states (biological or logical); it contains 6 SCCs with at least 2 nodes, the largest being composed of 10238 nodes and the others between 20 and 150.

Table 2 shows the results of a prototype implementation[2] of Algorithm 1 for the search of up to the 6-sets of biological component local states. One can observe that the execution time grows very rapidly with N compared to the number of visited nodes. This can be explained by intermediate nodes having a large set of cut N-sets leading to a costly computation of products.

While the precise biological interpretation of identified N-sets is out of the scope of this paper, we remark that the order of magnitude of the number of cut sets can be very different (more than 1000 cut 6-sets for SNAIL; none cut 6-sets for p21CIP1, except the gene that produces this protein). It supports a notion of robustness for the reachability of components, where the less cut sets, the more robust the reachability to various perturbations.

Applied to the `whole_PID_AND` model, our algorithm find in general much more cut N-sets, due to the conjunctive interpretation. This brings a significant increase in the execution time: the search up to the cut 5-sets took 1h, and the 6-sets leads to an out-of-memory failure.

Table 2. Results for the computation of cut N-sets for 3 local states. For each N, only the number of additional N-sets is displayed.

Model	N	Visited nodes	Exec. time	Nb. of resulting N-sets		
				$SNAIL_1$	$p15INK4b_1$	$p21CIP1_1$
whole_PID_OR	1	29022	0.9s	1	1	1
	2	36602	1.6s	+6	+6	+0
	3	44174	5.4s	+0	+92	+0
	4	54322	39s	+30	+60	+0
	5	68214	8.3m	+90	+80	+0
	6	90902	2.6h	+930	+208	+0

[2] Implemented as part of the PINT software – `http://process.hitting.free.fr` Models and scripts available at `http://loicpauleve.name/cutsets.tbz2`

The very large number of involved components makes intractable the naive exact algorithm consisting in enumerating all possible N-sets candidates and verifying the concerned reachability using model-checking. Similarly, making such a model fit into other frameworks, such as CellNetAnalyser (see previous subsection) is a challenging task, and might be considered as future work.

5 Discussion

We presented a new method to efficiently compute cut sets for the reachability of a local state of a component within networks of finite automata from any state delimited by a provided so-called context. Those cut sets are sets of automata local states such that disabling the activity of all local states of a cut set guarantees to prevent the reachability of the concerned local state. Automata networks are commonly used to represent the qualitative dynamics of interacting biological systems, such as signalling networks. The computation of cut sets can then lead to propose potential therapeutic targets that have been formally identified from the model for preventing the activation of a particular molecule.

The proposed algorithm works by propagating and combining the cut sets of local states along a *Graph of Local Causality* (GLC), that we introduce here upon automata networks. A GLC relates the local states that are necessary to occur prior to the reachability of the concerned local state. Several constructions of a GLC are generally possible and depend on the semantics of the model. We gave an example of such a construction for automata networks. That GLC has a size polynomial in the total number of local states in the network, and exponential in the number of local states within one automaton. Note that the core algorithm for computing the cut sets only requires as input a GLC satisfying a soundness property that can be easily extended to discrete systems that are more general than the automata networks considered here.

The computed cut sets are an under-approximation of the complete cut sets as the GLC abstracts several dynamical constraints from the underlying concrete model: any computed cut sets is a superset of a concrete cut set (potentially equal), some can be missed. Our algorithm prevents a costly enumeration of the potential sets of candidates, and aims at being tractable on very large networks.

A prototype implementation of our algorithm has been successfully applied to the extraction of cut sets from a Boolean model of a biological system involving more than 9000 interacting components. To our knowledge this is the first attempt of such a dynamical analysis for such large biological models. We note that most of the computation time is due to products between large sets of cut N-sets. To partially address this issue, we use of prefix trees to represent set of sets on which we have specialized operations to stick to sets of N-sets (Appendix A[3]). There is still room for improvement as our prototype does not implement any caching or variable re-ordering.

The work presented in this paper can be extended in several ways, notably with a *posterior enlarging of the cut sets*. Because the algorithm computes the

[3] Available at `http://loicpauleve.name/CAV2013-SI.pdf`

cut N-sets for each node in the GLC, it is possible to construct *a posteriori* cut sets with a greater cardinality by chaining them. For instance, let $kps \in \mathbb{V}(a_i)$ be a cut N-set for the reachability of a_i, for each $b_j \in kps$ and $kps' \in \mathbb{V}(b_j)$, $(kps \setminus \{b_j\}) \cup kps'$ is a cut set for a_i. In our biological case study, this method could be recursively applied until cut sets are composed of states of automata only acting for the environmental input.

With respect to the defined computation of cut N-sets, one could also derive *static reductions* of the GLC. Indeed, some particular nodes and arcs of the GLC can be removed without affecting the final valuation of nodes. A simple example are nodes representing objectives having no solution: such nodes can be safely removed as they bring no candidate N-sets for parents processes. These reductions conduct to both speed-up of the proposed algorithm but also to more compact representations for the reachability causality.

In addition of providing potential targets to prevent the occurrence of some behaviours, it may be crucial to ensure that the modified system keep satisfying important dynamical properties, as it is tackled with constrained minimal cut sets [4,8]. Currently, such constraints could be verified a posteriori in order to filter the computed cut sets that break important dynamics requirements. Taking advantage of those constraints during the computation is hence an promising research direction for large-scale applications.

Acknowledgements. LP and GA acknowledge the partial support of the French National Agency for Research (ANR-10-BLANC-0218 BioTempo project). The work of LP was supported by the Swiss SystemsX.ch project. The work of HK was supported by the Swiss National Science Foundation, grant no. PP00P2 128503.

References

1. Bernardinello, L., De Cindio, F.: A survey of basic net models and modular net classes. In: Rozenberg, G. (ed.) APN 1992. LNCS, vol. 609, pp. 304–351. Springer, Heidelberg (1992)
2. Bernot, G., Cassez, F., Comet, J.P., Delaplace, F., Müller, C., Roux, O.: Semantics of biological regulatory networks. Electronic Notes in Theoretical Computer Science 180(3), 3–14 (2007)
3. Drabsch, Y., Ten Dijke, P.: TGF-β signalling and its role in cancer progression and metastasis. Cancer Metastasis Rev. 31(3-4), 553–568 (2012)
4. Hädicke, O., Klamt, S.: Computing complex metabolic intervention strategies using constrained minimal cut sets. Metabolic Engineering 13(2), 204–213 (2011)
5. Harel, D., Kupferman, O., Vardi, M.Y.: On the complexity of verifying concurrent transition systems. Information and Computation 173(2), 143–161 (2002)
6. Hinkelmann, F., Laubenbacher, R.: Boolean models of bistable biological systems. Discrete and Continuous Dynamical Systems - Series S 4(6), 1443–1456 (2011)
7. de Jong, H.: Modeling and simulation of genetic regulatory systems: A literature review. Journal of Computational Biology 9, 67–103 (2002)
8. Jungreuthmayer, C., Zanghellini, J.: Designing optimal cell factories: integer programming couples elementary mode analysis with regulation. BMC Systems Biology 6(1), 103 (2012)

9. Kauffman, S.A.: Metabolic stability and epigenesis in randomly connected nets. Journal of Theoretical Biology 22, 437–467 (1969)
10. Klamt, S., Gilles, E.D.: Minimal cut sets in biochemical reaction networks. Bioinformatics 20(2), 226–234 (2004)
11. Klamt, S., Saez-Rodriguez, J., Gilles, E.: Structural and functional analysis of cellular networks with cellnetanalyzer. BMC Systems Biology 1(1), 2 (2007)
12. Klamt, S., Saez-Rodriguez, J., Lindquist, J., Simeoni, L., Gilles, E.: A methodology for the structural and functional analysis of signaling and regulatory networks. BMC Bioinformatics 7(1) 7(1), 56 (2006)
13. Lee, W.S., Grosh, D.L., Tillman, F.A., Lie, C.H.: Fault tree analysis, methods, and applications - a review. IEEE Transactions on Reliability R-34, 194–203 (1985)
14. Moustakas, A., Heldin, C.H.: Signaling networks guiding epithelial-mesenchymal transitions during embryogenesis and cancer progression. Cancer Sci. 98(10), 1512–1520 (2007)
15. Paulevé, L., Magnin, M., Roux, O.: Refining dynamics of gene regulatory networks in a stochastic π-calculus framework. In: Priami, C., Back, R.-J., Petre, I., de Vink, E. (eds.) Transactions on Computational Systems Biology XIII. LNCS, vol. 6575, pp. 171–191. Springer, Heidelberg (2011)
16. Paulevé, L., Magnin, M., Roux, O.: Static analysis of biological regulatory networks dynamics using abstract interpretation. Mathematical Structures in Computer Science 22(4), 651–685 (2012)
17. Richard, A.: Negative circuits and sustained oscillations in asynchronous automata networks. Advances in Applied Mathematics 44(4), 378–392 (2010)
18. Samaga, R., Saez-Rodriguez, J., Alexopoulos, L.G., Sorger, P.K., Klamt, S.: The logic of egfr/erbb signaling: Theoretical properties and analysis of high-throughput data. PLoS Comput. Biol. 5(8), e1000438 (2009)
19. Samaga, R., Von Kamp, A., Klamt, S.: Computing combinatorial intervention strategies and failure modes in signaling networks. Journal of Computational Biology 17(1), 39–53 (2010)
20. Schaefer, C.F., Anthony, K., Krupa, S., Buchoff, J., Day, M., Hannay, T., Buetow, K.H.: PID: The Pathway Interaction Database. Nucleic Acids Res. 9, D674–D679 (2009)
21. Shier, D.R., Whited, D.E.: Iterative algorithms for generating minimal cutsets in directed graphs. Networks 16(2), 133–147 (1986)
22. Tang, Z., Dugan, J.: Minimal cut set/sequence generation for dynamic fault trees. In: Reliability and Maintainability, 2004 Annual Symposium, RAMS (2004)
23. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM Journal on Computing 1(2), 146–160 (1972)
24. Thomas, R.: Boolean formalization of genetic control circuits. Journal of Theoretical Biology 42(3), 563–585 (1973)

Model-Checking Signal Transduction Networks through Decreasing Reachability Sets

Koen Claessen[1], Jasmin Fisher[2], Samin Ishtiaq[2], Nir Piterman[3], and Qinsi Wang[4]

[1] Chalmers University
[2] Microsoft Research Cambridge
[3] University of Leicester
[4] Carnegie Mellon University

Abstract. We consider model checking of *Qualitative Networks*, a popular formalism for modeling signal transduction networks in biology. One of the unique features of qualitative networks, due to them lacking initial states, is that of "reducing reachability sets". Simply put, a state that is not visited after i steps will not be visited after i' steps for every $i' > i$. We use this feature to create a compact representation of all the paths of a qualitative network of a certain structure. Combining this compact path representation with LTL model checking leads to significant acceleration in performance. In particular, for a recent model of Leukemia, our approach works at least 5 times faster than the standard approach and up to 100 times faster in some cases. Our approach enhances the iterative hypothesis-driven experimentation process used by biologists, enabling fast turn-around of executable biological models.

1 Introduction

Formal verification methods hold great promise for biological research. Over the years, various efforts have repeatedly demonstrated that the use of formal methods is beneficial for gaining new biological insights as well as directing new experimental avenues. Experimental biology is an iterative process of hypothesis-driven experimentation of a particular biological system of interest. The idea to boost this process using formal executable models describing aspects of biological behaviors, also known as Executable Biology, has proved to be successful in cell biology and shed new light on cell signalling and cell-cell communication.

Biological systems can be modelled at different levels of abstraction. On a cellular level, we usually consider the causality relations between molecular species (e.g., genes, proteins) inside the cell (collectively called signal transduction networks), and between cells (intercellular signalling). These can be described by various state-transition systems such as compositional state machines, Petri nets, and process calculi.

One successful approach to the usage of abstraction in biology has been the usage of Boolean networks [23]. Boolean networks call for abstracting the status of each modeled substance as either active (on) or inactive (off). Although a

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 85–100, 2013.

very high level abstraction, it has been found useful to gain better understanding of certain biological systems [20, 22]. The appeal of this discrete approach along with the shortcomings of the very aggressive abstraction, led researchers to suggest various formalisms such as Qualitative Networks [21] and Gene Regulatory Networks [16] that allow to refine models when compared to the Boolean approach. In these formalisms, every substance can have one of a small discrete number of levels. Dependencies between substances become algebraic functions instead of Boolean functions. Dynamically, a state of the model corresponds to a valuation of each of the substances and changes in values of substances occur gradually based on these algebraic functions. Qualitative networks and similar formalisms (e.g., genetic regulatory networks [23]) have proven to be a suitable formalism to model some biological systems [3, 20, 21, 23].

Here, we consider model checking of qualitative networks. One of the unique features of qualitative networks is that they have no initial states. That is, the set of initial states is the set of all states. Obviously, when searching for specific executions or when trying to prove a certain property we may want to restrict attention to certain initial states. However, the general lack of initial states suggests a unique approach towards model checking. It follows that if some state is not reachable by exactly n steps, for some n, it will not be reachable by exactly n' steps, for every $n' > n$. These "decreasing" sets of reachable states allow to create a more efficient symbolic representation of all the paths of a certain length.

However, this observation alone is not enough to create an efficient model checking procedure. Indeed, accurately representing the set of reachable states at a certain time amounts to the original problem of model checking (for reachability), which does not scale. In order to address this we use an over-approximation of the set of states that are reachable by exactly n steps. We represent the over-approximation as a Cartesian product of the set of values that are reachable for each variable at every time point. The computation of this over-approximation never requires us to consider more than two adjacent states of the system. Thus, it can be computed quite efficiently. Then, using this over-approximation we create a much smaller encoding of the set of possible paths in the system.

Finally, an LTL formula is translated to an additional set of constraints on the set of paths. Our encoding is based on temporal testers [17].

We test our implementation on many of the biological models developed using Qualitative Networks. The experimental results show that there is significant acceleration when considering the decreasing reachability property of qualitative networks. In many examples, in particular larger and more complicated biological models, this technique leads to considerable speedups. The technique scales well with increase of size of models and with increase in length of paths sought for.

These results are especially encouraging given the methodology biologists have been using when employing our tools [2]. Typically, models are constructed and then compared with experimental results. The process of model development is a highly iterative process involving trial and error where the biologist compares a current approach with experimental results and refines the model until it matches current experimental knowledge. In this iterative process it is important to give

fast answers to queries of the biologist. We hope that with the speed ups afforded by this new technique, model checking could be incorporated into the routine methodology of experimental biologists using our tools.

1.1 Related Work

Over the last decade, the usage of model checking has proven to be extremely useful for the analysis of discrete biological models (e.g., [4,5,9,12,13,21]), leading to numerous new biological findings. This is part of a general "trend" to adopt (and adapt) approaches from formal verification and formal methods for usage in biological modeling [1, 11, 12, 14].

Analysis of Boolean Networks and Qualitative Networks has been done either manually or with ad-hoc techniques. The main form of analysis applied is that of stability. That is, identifying in which states the system can remain even after very long executions. In particular, if all states of the system (or most of them) lead to the same loop and if that loop is small (in particular if it consists of one state) then the network is more "stable". Traditionally, analysis was done by manually inspecting the graphs of configurations that the system gives rise to [10, 15, 19].

Obviously, this approach severely limits the size of systems that can be analyzed. As qualitative networks define finite-state systems, the technology to support model checking through existing model checkers is widely available. Previous attempts to use standard model checking tools and techniques on these types of biological models have proven unsatisfactory [8]. Both SAT-based and BDD-based techniques could not scale to handle large models.

In previous work (*e.g.* [21], [8]) custom techniques have been developed for reasoning about the *final states* reached by biological models. The technique in [8] is related to our results as it uses a similar abstraction domain to reason about sets of states of a Qualitative Network. However, the approach in [8] is specifically tailored for stabilization and it was not clear how to apply it to model checking (or path representation). It is also more aggressive than the approach advocated here and scales to larger models.

In [18], the techniques in [8] are used for reasoning about hardware designs. They show how to use these techniques to handle a restricted subset of LTL.

1.2 Structure of the Paper

The paper is structured as follows. In Section 2 we introduce qualitative networks and their usage through an example. In Section 3 we give the formal background and fix notations. Then, in Section 4 we explain the decreasing reachability concept and the abstraction that we use with it. We also include a brief explanation of the types of paths we are looking for. We then give some experimental results (Section 5). We conclude in Section 6.

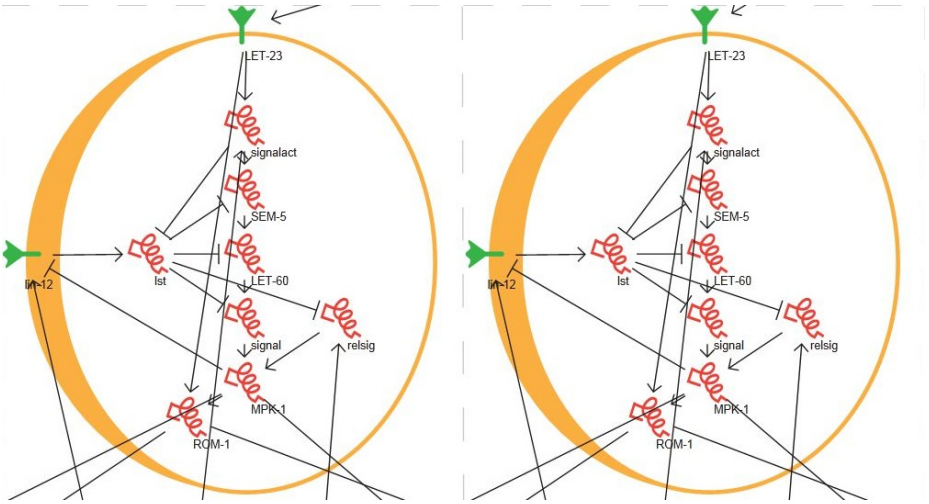


Fig. 1. A pictorial view of part of a model describing aspects of cell-fate determination during *C. elegans* vulval development [12]. The image shows two cells having the "same program". Neighboring cells and connections between cells are not shown.

2 Qualitative Networks Example

We start with an example giving some introduction to Qualitative Networks and the usage of LTL model checking in this context.

Figure 1 shows a model representing aspects of cell-fate determination during *C. elegans* vulval development [12]. The part shown in the figure includes two cells. Each cell in the model represents a *vulval precursor cell* and the elements inside it represent proteins whose level of activity influences the decision of the cell as to which part of the vulva the descendants of the cell should form. All cells execute the same program and it is the communication between the cells themselves as well as communication between the cells and additional parts of the model (i.e., external signals) that determine a different fate for each of the cells. Understanding cell-fate determination is crucial for our understanding of normal development processes as well as occasions where these go wrong such as disease and cancer. The pictorial view gives rise to a formal model expressed as a qualitative network [21]. Formal definitions are in the next section.

Each of the cells in the model includes executing components, for example `LET-60`, that correspond to a single variable. Each variable v holds a value, which is a number in $\{0, 1, \ldots, N_v\}$, where N_v is the *granularity* of the variable. Specifically, in Figure 1 all variables range over $\{0, 1, 2\}$. A *target function*, T_v, defined over the values of variables affecting v (i.e., having incoming arrows into v), determines how v is updated: if $v < T_v$ and $v < N_v$ then $v' = v + 1$, if $v > T_v$ and $v > 0$ then $v' = v - 1$, else v does not change. In a qualitative network all variables are updated synchronously in parallel.

Intuitively, the update function of each variable is designed such that the value of the variable follows its target function, which depends on other variables. In the biological setting, the typical target of a variable, v, combines the positive influence of variables $w_1, w_2, \ldots w_s$ with the negative influence of variables $w_{s+1}, w_{s+2}, \ldots w_{s+r}$:

$$T_v(w_1, w_2, \ldots w_{s+r}) = \max\left(0, \left\lfloor \frac{1}{s}\sum_{k=1}^{s} w_k - \frac{1}{r}\sum_{k=1}^{r} w_{s+k} + \frac{1}{2} \right\rfloor\right)$$

Graphically, this is often represented as an *influence graph* with $\rightarrow$ edges between each of $w_1, w_2, \ldots, w_s$ and v and $\multimap$ edges between each of $w_{s+1}, w_{s+2}, \ldots, w_{s+r}$ and v. More complicated target functions can be defined using algebraic expressions over $\{w_1, \ldots, w_{s+r}\}$. We refer the reader to [2, 3, 20, 21] for further details about other modeling options.

Specifically, in the model above, the target of `lst` is:

$$T_{\texttt{lst}} = min(2 - \texttt{signalact}, 1) * \texttt{lin-12}$$

This models activation by `lin-12` and inhibition by `signalact`. However, inhibition occurs only when `signalact` is at its maximal level (2). When inhibition is not maximal the target follows the value of `lin-12`. The target of `SEM-5` is:

$$T_{\texttt{SEM-5}} = max(0, 2 - ((2 - \texttt{signalact}) * (max(\texttt{lst} - 1, 0) + 1)))$$

This function means that `lst` inhibits `SEM-5` and `signalact` activates it. However, activation takes precedence: inhibition takes effect only in case that activation is not at its maximum value (2), and only when inhibition is at its maximum value (2). Otherwise, the target follows the value of its activator (`signalact`).

Models are analyzed to ensure that they reproduce behavior that is observed in experiments. A mismatch between the model and experimental observations signifies that something is wrong with our understanding of the system. In such a case, further analysis is required in order to understand whether and how the model needs to be changed. Models are usually analyzed by simulating them and following the behavior of components. A special property of interest in these types of models is that of *stability*: there is a unique state that has a self loop and all executions lead to that state [8, 21]. When a model does stabilize it is interesting to check the value of variables in the stabilization point. In addition, regardless of whether the model is stabilizing or not, model checking is used to prove properties of the model or to search for interesting executions. For the model in Figure 1 the following properties, e.g., are of interest.

– Do there exist executions leading to adjacent primary fates in which increase of LS happens after down-regulation of `lin-12`?
  This property is translated to an LTL formula of the following format:

$$\theta \wedge FGf_{i,j} \wedge (\neg d_i U l_i) \wedge (\neg d_j U l_i),$$

  where θ is some condition on initial states, $f_{i,j}$ is the property characterizing the states in which VPCs i and j are both in primary fate, d_i is the property

that `lin-12` is low in VPC i, l_i is the property that d_i is high in VPC i, and d_j and l_j are similar for VPC j. This property is run in positive mode, i.e., we are searching for execution that satisfies this property.

– Is it true that for runs starting from a given set of states the sequence of occurrences leading to fate execution follows the pattern: MPK-1 increases to high level then `lin-12` is down-regulated, and then LS is activated.
This property is translated to an LTL formula of the following format:

$$\theta \implies F(m_i \wedge XF(l_i \wedge XFd_i)),$$

where θ is some condition on initial states, m_i is the property characterizing states in which VPC i has a high level of MPK-1, l_i is the property characterizing states in which VPC i has a low level of `lin-12`, and d_i is the property characterizing states in which VPC i has a high level of LS. This property is run in negative (model checking) mode, i.e., we are searching for executions falsifying this property and expecting the search to fail.

3 Formal Background

We formally introduce the Qualitative Networks (QN for short) framework and recall the definition of linear temporal logic (LTL for short).

Following [21], a *qualitative network* (QN), $Q(V,T,N)$, of granularity $N+1$ consists of variables: $V = (v_1, v_2 \dots v_n)$.[1] A state of the system is a finite map $s : V \to \{0,1,\dots N\}$. Each variable $v_i \in V$ has a *target function* $T_i \in T$ associated with it: $T_i{:}\{0,1,\dots N\}^n \to \{0,1,\dots N\}$. Qualitative networks update the variables using synchronous parallelism.

Target functions in qualitative networks direct the execution of the network: from state $s = (d_1, d_2 \dots d_n)$, the *next state* $s' = (d'_1, d'_2 \dots d'_n)$ is computed by:

$$d'_i = \begin{cases} d_i + 1 & d_i < T_i(s) \text{ and } d_i < N, \\ d_i - 1 & d_i > T_i(s) \text{ and } d_i > 0, \\ d_i & \text{otherwise} \end{cases} \tag{1}$$

A target function of a variable v is typically a simple algebraic function, such as sum, over several other variables $w_1, w_2 \dots w_m$ (see, e.g., Section 2). We often say that v *depends* on $w_1, w_2 \dots w_m$ or that $w_1, w_2 \dots w_m$ are *inputs* of v. In the following, we use the term *network* to refer to a qualitative network.

A QN $Q(V,T,N)$ defines a state space $\Sigma = \{s : V \to \{0,1,\dots N\}\}$ and a transition function $f : \Sigma \to \Sigma$, where $f(s) = s'$ such that for every $v \in V$ we have $s'(v)$ depends on $T_v(s)$ as in Equation (1). For a state $s \in \Sigma$ we denote by $s(v)$ also by s_v. In particular, $f_v(s) = f(s)(v)$ is the value of v in $f(s)$. We say that a state s is *recurring* if it is possible to get back to s after a finite number of applications of f. That is, if for some $i > 0$ we have $f^i(s) = s$. As the state space

[1] For simplicity, we assume that all variables have the same range $\{0,\dots,N\}$. The extension to individual ranges is not complicated. Our implementation supports individual ranges for variables.

of a qualitative network is finite, the set of recurring states is never empty. We say that a network is *stabilizing* if there exists a unique recurring state s. That is, there is a unique state s such that $f(s) = s$, and for every other state s' and every $i > 0$ we have $f^i(s') \neq s'$. Intuitively, this means that starting from an arbitrary state, we always end up in a fixpoint and always the same one. A run of a QN $Q(V, T, N)$ is an infinite sequence $r = s_0, s_1, \ldots$ such that for every $i \geq 0$ we have $s_i \in \Sigma$ and $s_{i+1} = f(s_i)$.

We define linear temporal logic (LTL) over runs of qualitative networks as follows. For every variable $v \in V$ and every value $n \in \{0, 1, \ldots N\}$, we define an atomic proposition $v \bowtie n$, where $\bowtie \in \{>, \geq, \leq, <\}$. Let AP denote the set of all atomic propositions (for a network Q). The set of LTL formulas is:

$$\varphi ::= \mathsf{AP} \mid \varphi \vee \varphi \mid \neg\varphi \mid X\varphi \mid \varphi U \varphi$$

As usual, we introduce $\wedge$, $\rightarrow$, F, and G, as syntactic sugar.

An LTL formula φ is satisfied over a run $r = s_0, s_1, \ldots$ in location i, denoted $r, i \vDash \varphi$ according to the following:

- For $\varphi = v \bowtie n \in \mathsf{AP}$ we have $r, i \vDash \varphi$ if $s_i(v) \bowtie n$.
- For $\varphi = \neg\psi$ we have $r, i \vDash \varphi$ if it is not the case that $r, i \vDash \psi$.
- For $\varphi = \psi_1 \vee \psi_2$ we have $r, i \vDash \psi_1 \vee \psi_2$ if either $r, i \vDash \psi_1$ or $r, i \vDash \psi_2$.
- For $\varphi = X\psi$ we have $r, i \vDash \varphi$ if $r, i + 1 \vDash \psi$.
- For $\varphi = \psi_1 U \psi_2$ we have $r, i \vDash \varphi$ if there is $j \geq i$ such that $r, j \vDash \psi_2$ and for every $i \leq k < j$ we have $r, k \vDash \psi_1$.

We say that a run r satisfies an LTL formula φ, denoted $r \vDash \varphi$ if $r, 0 \vDash \varphi$. Given a Qualitative Network Q, we say that Q satisfies an LTL formula φ, denoted $Q \vDash \varphi$, if for every run r of Q we have $r \vDash \varphi$. In case that $Q \nvDash \varphi$ a *counter example* is a run r such that $r \nvDash \varphi$.

We use bounded model checking [7] for checking whether a qualitative network satisfies a given LTL formula φ. Intuitively, we search for a run of a certain structure (and length) that does not satisfy the formula by constructing a Boolean formula whose satisfiability corresponds to such a run. Searching for a counter example of length l means that we (1) create Boolean variables that represent the state of the system in l different time points, (2) add constraints that enforce that the transition of the qualitative network holds between every two consecutive time points, (3) add constraints that enforce that the transition of the qualitative network holds between the state at time $l - 1$ (last state) and some previous state (i.e., that the sequence of states ends in a loop), and (4) add Boolean variables and constraints that enforce satisfaction of the (negation of) the temporal property.

In order to create a Boolean encoding of the LTL formula we use a variant of the temporal testers approach in [17]. Specifically, for every temporal subformula (and every time point in the trace) we add a Boolean variable that tracks the truth value of the subformula at that time. The truth value of these variables are connected to the truth values of propositions (encoded through the state of the model) and truth values of other subformulas. In addition, we add constraints

```
Concrete Decreasing Reachability
1 Σ_0 = Σ;
2 Σ_{-1} = ∅;
3 j := 0;
4 while (Σ_{j-1} ≠ Σ_j) {
5     Σ_{j+1} = Σ_j ∖ {s' ∈ Σ_j | ∀s ∈ Σ_j.s' ≠ f(s)};
6     j++;
7 }
8 return Σ_0, ..., Σ_{j-1};
```

Fig. 2. Computing decreasing reachability sets

that enforce satisfaction of eventualities in the loop. In order to search for a trace that satisfies a certain LTL formula we add the encoding of the formula to the trace. Satisfiability then provides a run satisfying the formula. To prove that all runs up of a certain length satisfy a formula, we add the encoding of the negation of the formula to the trace. Unsatisfiability then provides a proof that no run (of the given length) satisfies the formula.

4 Decreasing Reachability Sets

A notable difference between QNs and "normal" transition systems is that QNs do not specify initial states. For example, for the classical stability analysis all states are considered as initial states. It follows that if a state s of a QN is not reachable after i steps, it is not reachable after i' steps for every $i' > i$. Thus, there is a decreasing sequence of sets $\Sigma_0 \supseteq \Sigma_1 \supseteq \cdots \supseteq \Sigma_l$ such that searching for runs of the network can be restricted to the set of runs of the form $\Sigma_0 \cdot \Sigma_1 \cdots (\Sigma_l)^\omega$. Here we show how to take advantage of this fact in constructing a more scalable model checking algorithm for qualitative networks.

Consider a Qualitative Network $Q(V, T, N)$ with set of states $\Sigma : V \to \{0, \ldots, N\}$. We say that a state $s \in \Sigma$ is reachable by exactly i steps if there is some run $r = s_0, s_1, \ldots$ such that $s = s_i$. Dually, we say that s is not reachable by exactly i steps if for every run $r = s_0, s_1, \ldots$ we have $s_i \neq s$.

Lemma 1. *If a state s is not reachable by exactly i steps then it is not reachable by exactly i' steps for every $i' > i$.*

The algorithm in Figure 2 computes a decreasing sequence $\Sigma_0 \supset \Sigma_1 \supset \cdots \supset \Sigma_{j-1}$ such that all states that are reachable by exactly i steps are in Σ_i if $i < j$ and in Σ_{j-1} if $i \geq j$. We note that the definition of Σ_{j+1} in line 5 is equivalent to the standard $\Sigma_{j+1} = f(\Sigma_j)$. However, we choose to write it as in the algorithm above in order to stress that only states in Σ_j are candidates for inclusion in Σ_{j+1}. Given the sets $\Sigma_0, \ldots, \Sigma_{j-1}$ every run $r = s_0, s_1, \ldots$ of Q satisfies $s_i \in \Sigma_i$ for $i < j$ and $s_i \in \Sigma_{j-1}$ for $i \geq j$. In particular, if $Q \not\models \varphi$ for some LTL formula φ then the run witnessing the unsatisfaction of φ can be searched for in this smaller space of runs. Unfortunately, the algorithm in Figure 2 is not feasible. Indeed, it amounts to computing the exact reachability sets of the QN Q, which does not scale well [8].

```
ABSTRACT DECREASING REACHABILITY
1  ∀v_i ∈ V . D_{i,0} := {0, 1, ... N};
2  ∀v_i ∈ V . D_{i,-1} := ∅;
3  j := 0;
4  while (∃v_i ∈ V . D_{i,j} ≠ D_{i,j-1}) {
5      foreach (v_i ∈ V) {
6          D_{i,j+1} := ∅;
7          foreach (d ∈ D_{i,j}){
8              if (∃(d_1, ..., d_m) ∈ D_{1,j} × ··· × D_{m,j} . f_v(d_1, ..., d_m) = d){
9                  D_{i,j+1} := D_{i,j+1} ∪ {d};
10             }
11         }
12     }
13     j++;
14 }
15 return ∀v_i ∈ V . ∀j' ≤ j . D_{i,j'};
```

Fig. 3. Over-approximating decreasing reachability sets

In order to effectively use Lemma 1 we combine it with over-approximation, which leads to a scalable algorithm. Specifically, instead of considering the set Σ_k of states reachable at step k, we identify for every variable $v_i \in V$ the domain $D_{i,k}$ of the set of values possible at time k for variable v_i. Just like the general set of states, when we consider the possible values of variable v_i we get that $D_{i,0} \supseteq D_{i,1} \supseteq \cdots \supseteq D_{i,l}$. The advantage is that the sets $D_{i,k}$ for all $v_i \in V$ and $k > 0$ can be constructed by induction by considering only the knowledge on previous ranges and the target function of one variable.

Consider the algorithm in Figure 3. For each variable, it initializes the set of possible values at time 0 as the set of all values. Then, based on the possible values at time j it computes the possible values at time $j+1$. The actual check can be either implemented explicitly if the number of inputs of all target functions is small (as in most cases) or symbolically (see [6]). Considering only variables (and values) that are required to decide the possible values of variable v_i at time j makes the problem much smaller than the general reachability problem. Notice that, again, only values that are possible at time j need be considered at time $j+1$. That is, $D_{i,j+1}$ starts as empty (line 6) and only values from $D_{i,j}$ are added to it (lines 7–10). As before, $D_{i,j+1}$ is the projection of $f(D_{1,j} \times \ldots \times D_{m,j})$ on v_i. The notation used in the algorithm above stresses that only states in $D_{i,j}$ are candidates for inclusion in $D_{i,j+1}$.

The algorithm produces very compact information that enables to follow with a search for runs of the QN. Namely, for every variable v_i and for every time point $0 \leq k < j$ we have a decreasing sequence of domains

$$D_{i,0} \supseteq D_{i,1} \supseteq \cdots \supseteq D_{i,k}.$$

Consider a Qualitative Network $Q(V,T,N)$, where $V = \{v_1, \ldots, v_n\}$ and a run $r = s_0, s_1, \ldots$. As before, every run $r = s_0, s_1, \ldots$ satisfies that for every i and for every t we have $s_t(v_i) \in D_{i,t}$ for $t < j$ and $s_t(v_i) \in D_{i,j-1}$ for $t \geq j$.

We look for paths that are in the form of a lasso, as we explain below. We say that r is a *loop of length* l if for some $0 < k \leq l$ and for all $m \geq 0$ we have $s_{l+m} = s_{l+m-k}$. That is, the run r is obtained by considering a prefix of length $l-k$ of states and then a loop of k states that repeats forever. A search for a loop of length l that satisfies an LTL formula φ can be encoded as a bounded model checking query as follows. We encode the existence of l states $s_0, \ldots, s_{l-1}$. We use the decreasing reachability sets $D_{i,t}$ to force state s_t to be in $D_{0,t} \times \cdots \times D_{n,t}$. This leads to a smaller encoding of the states $s_0, \ldots, s_{l-1}$ and to smaller search space. We add constraints that enforce that for every $0 \leq t < l-1$ we have $s_{t+1} = f(s_t)$. Furthermore, we encode the existence of a time $l-k$ such that $s_{l-k} = f(s_{l-1})$. We then search for a loop of length l that satisfies φ. It is well known that if there is a run of Q that satisfies φ then there is some l and a loop of length l that satisfies φ. We note that sometimes there is a mismatch between the length of loop sought for and length of sequence of sets (j) produced by the algorithm in Figure 3. Suppose that the algorithm returns the sets $D_{i,t}$ for $v_i \in V$ and $0 \leq t < j$. If $l > j$, we use the sets $D_{i,j-1}$ to "pad" the sequence. Thus, states $s_j, \ldots, s_{l-1}$ will also be sought in $\Pi_i D_{i,j-1}$. If $l < j$, we use the sets $D_{i,0}, \ldots, D_{i,l-2}, D_{i,j-1}$ for $v_i \in V$. Thus, only the last state s_{l-1} is ensured to be in our "best" approximation $\Pi_i D_{i,j-1}$. A detailed explanation of how we encode the decreasing reachability sets as a Boolean satisfiability problem is given in [6].

5 Experimental Results

We implemented this technique to work on models defined through our tool BMA [2]. Here, we present experimental results of running our implementation on a set of benchmark problems. We collected a total of 22 benchmark problems from various sources (skin cells differentiation models from [8,21], diabetes models from [3], models of cell fate determination during *C. elegans* vulval development, a *Drosophila* embryo development model from [20], Leukemia models constructed by ourselves, and a few additional examples constructed by ourselves). The number of variables in the models and the maximal range of variables is reported in Table 1.

Our experiments compare two encodings. First, the encoding explained in Section 4, referred to as *opt* (for optimized). Second, the encoding that considers l states $s_0, \ldots, s_l$ where $s_t(v_i) \in \{0, \ldots, N\}$ for every t and every i. That is, in terms of the explanation in Section 4 for every variable v_i and every time point $0 \leq t \leq l$ we consider the set $D_{i,t} = \{0, \ldots, N\}$. This encoding is referred to as *naïve*. In both cases we use the same encoding to a Boolean satisfiability problem. Further details about the exact encoding can be found in [6].

We performed two kinds of experiments. First, we search for loops of length 10, 20, ···, 50 on all the models for the optimized and naïve encodings. Second, we search for loops that satisfy a certain LTL property (either as a counter example to model checking or as an example run satisfying a given property). Again, this is performed for both the optimized and the naïve encodings. LTL properties are considered only for four biological models. The properties were suggested by our collaborators as interesting properties to check for these models.

Table 1. Number of variables in models and their ranges

Model name	♯Vars	Range	Model name	♯Vars	Range
2var_unstable	2	0..1	Bcr-Abl	57	0..2
Bcr-AblNoFeedbacks	54	0..2	BooleanLoop	2	0..1
NoLoopFound	5	0..4	Skin1D_TF_0	75	0..4
Skin1D_TF_1	75	0..4	Skin1D	75	0..4
Skin2D_3cells_2layers_0	90	0..4	Skin2D_3cells_2layers_1	90	0..4
Skin2D_3cells_2layers_2	90	0..4	Skin2D_5X2_TF	198	0..4
Skin2D_5X2	198	0..4	SmallTestCase	3	0..4
SSkin1D_0	30	0..4	SSkin1D_TF_1	31	0..4
SSkin1D	30	0..4	SSkin2D_3cells_2layers	40	0..4
VerySmallTestCase	2	0..4	VPC_lin15ko	85	0..2
VPC_Non_stabilizing	33	0..2	VPC_stabilizing	43	0..2

For both experiments, we report separately on the global time and the time spent in the SAT solver. All experiments were run on an Intel Xeon machine with CPU X7560@2.27GHz running Windows Server 2008 R2 Enterprise.

In Tables 2 and 3 we include experimental results for the search for loops. We compare the global run time of the optimized search vs the naïve search. The global run time for the optimized search includes the time it takes to compute the sequence of decreasing reachability sets. Accordingly, in some of the models, especially the smaller ones, the overhead of computing this additional information makes the optimized computation slower than the naïve one. For information we include also the net runtime spent in the SAT solver.

In Table 4 we include experimental results for the model checking experiment. As before, we include the results of running the search for counter example of lengths 10, 20, 30, 40, and 50. We include the total runtime of the optimized vs the naïve approaches as well as the time spent in the SAT solver. As before, the global runtime for the optimized search includes the computation of the decreasing reachability sets. The properties in the table are of the following form. Let I, $a - d$ denote formulas that are Boolean combinations of propositions.

- $I \rightarrow (\neg a)Ub$ – we check that the sequence of events when starting from the given initial states (I)satisfies the order b happens before a.
- $I \wedge FGa \wedge F(b \wedge XFc)$ – we check that the model gets from some states (I) to a loop that satisfies the condition a and the path leading to the loop satisfies that b happens first and then c.
- $I \wedge FGa \wedge F(b \wedge XF(c \wedge XFd))$ – we extend the previous property by checking the sequence b then c then d.
- $I \wedge FGa \wedge (\neg b)Uc$ – we check that the model gets from some states (I) to a loop that satisfies the condition a and the path leading to the loop satisfies that b cannot happen before c.
- $GFa \wedge GFb$ – we check for the existence of loops that exhibit a form of instability by having states that satisfy both a and b.

When considering the path search, on many of the smaller models the new technique does not offer a significant advantage. However, on larger models, and

Table 2. Searching for loops (10, 20, 30)

Length of loop	10				20				30			
	Global Time (Seconds)		Sat Time (Seconds)		Global Time (Seconds)		Sat Time (Seconds)		Global Time (Seconds)		Sat Time (Seconds)	
Model name	Naïve	Opt	Naïve	Opt	Naïve	Opt	Naïve	Opt	Naïve	Opt	Naïve	Opt
2var_unstable	6.92	0.78	0.21	0	0.46	0.54	0	0	0.51	0.57	0	0
Bcr-Abl	67.76	9.32	28.92	1.46	196.68	9.49	142.41	1.31	281.27	10.29	108.14	1.85
Bcr-AblNoFeedbacks	66.52	6.77	29.58	0.71	201.59	6.71	101.69	0.56	307.60	6.60	219.72	0.62
BooleanLoop	0.49	0.51	0	0	0.48	0.57	0.01	0	0.53	0.59	0.01	0.01
NoLoopFound	0.78	0.74	0.06	0.01	1.14	0.93	0.09	0.03	1.45	1.04	0.10	0.06
Skin1D_TF_0	136.21	140.78	122.85	127.47	218.52	80.33	191.06	55.23	127.28	96.49	86.05	60.06
Skin1D_TF_1	167.32	173.03	154.00	159.55	698.47	445.32	670.77	419.24	883.35	572.03	842.06	536.04
Skin1D	90.92	68.82	77.63	54.54	45.67	23.21	17.55	8.77	133.72	23.46	92.36	8.13
Skin2D_3cells_2layers_0	567.31	640.71	545.49	618.44	238.28	205.15	192.28	162.14	164.79	218.77	93.45	153.11
Skin2D_3cells_2layers_1	910.08	553.27	891.70	535.02	82.04	117.48	44.70	82.79	122.77	219.04	64.96	167.65
Skin2D_3cells_2layers_2	315.20	169.92	293.45	151.64	121.12	36.58	74.49	18.74	188.78	39.36	114.81	20.15
Skin2D_5X2_TF	511.31	223.93	459.38	182.65	1466.90	391.96	1378.80	353.06	1275.30	73.77	1135.25	35.83
Skin2D_5X2	343.96	85.64	300.03	56.71	721.58	57.20	630.92	28.46	965.24	48.26	828.12	16.83
SmallTestCase	0.53	0.54	0.01	0	0.54	0.73	0.01	0	0.60	0.54	0.01	0
SSkin1D_0	70.71	69.00	63.71	61.93	21.35	20.71	5.87	5.93	33.07	32.74	12.52	12.34
SSkin1D_TF_1	9.77	10.05	2.88	2.93	22.85	26.02	8.23	9.04	35.61	35.16	15.12	14.96
SSkin1D	145.28	146.74	138.61	139.76	32.00	33.38	18.29	18.51	33.89	33.80	13.57	13.49
SSkin2D_3cells_2layers	301.33	158.62	286.80	148.08	63.46	50.12	35.44	36.14	86.26	32.41	44.30	14.91
VerySmallTestCase	0.37	0.42	0	0	0.39	0.43	0.01	0	0.40	0.43	0.01	0
VPC_lin15ko	8.31	6.81	3.35	0.32	14.87	6.74	5.13	0.26	21.99	6.76	7.42	0.20
VPC_Non_stabilizing	3.43	3.40	0.85	0.26	6.02	3.95	1.23	0.29	9.35	4.87	2.10	0.62
VPC_stabilizing	3.31	4.79	0.74	0.14	5.84	4.79	0.99	0.18	9.10	4.67	1.92	0.14

Table 3. Searching for loops (40, 50)

Length of loop	40				50			
	Global Time (Seconds)		Sat Time (Seconds)		Global Time (Seconds)		Sat Time (Seconds)	
Model name	Naïve	Opt	Naïve	Opt	Naïve	Opt	Naïve	Opt
2var_unstable	0.54	0.60	0.01	0	1.05	0.64	0.01	0.01
Bcr-Abl	667.22	11.54	552.90	2.74	1019.68	11.94	869.56	2.76
Bcr-AblNoFeedbacks	574.61	6.79	316.07	0.64	857.17	6.90	719.21	0.69
BooleanLoop	0.54	0.60	0.01	0.01	0.59	0.66	0.01	0.01
NoLoopFound	1.90	1.15	0.23	0.04	2.23	1.34	0.22	0.05
Skin1D_TF_0	126.13	153.85	68.31	104.11	224.38	247.93	149.55	182.58
Skin1D_TF_1	108.84	160.72	52.01	112.33	167.86	290.97	91.13	228.46
Skin1D	122.73	29.39	64.99	12.84	259.75	34.04	182.50	16.09
Skin2D_3cells_2layers_0	391.08	325.43	293.83	237.89	470.89	663.87	341.24	545.49
Skin2D_3cells_2layers_1	196.99	271.98	118.22	202.01	476.94	557.09	366.61	464.88
Skin2D_3cells_2layers_2	413.13	44.06	314.95	23.75	445.78	47.51	308.71	25.18
Skin2D_5X2_TF	3067.08	93.15	2649.01	48.12	5135.87	82.38	3956.13	34.25
Skin2D_5X2	2403.53	47.69	2149.43	14.87	4025.83	56.86	3254.90	18.18
SmallTestCase	0.96	0.57	0.02	0	0.77	0.58	0.02	0
SSkin1D_0	44.81	42.03	13.52	13.37	58.09	57.45	22.64	21.91
SSkin1D_TF_1	43.97	46.26	15.88	16.13	60.46	60.52	22.77	24.35
SSkin1D	41.13	41.49	12.48	12.59	60.77	61.34	22.82	22.87
SSkin2D_3cells_2layers	117.64	42.86	50.82	20.36	157.07	51.19	80.54	22.95
VerySmallTestCase	0.48	0.44	0	0	0.81	0.67	0.01	0
VPC_lin15ko	27.04	6.94	7.34	0.20	45.70	7.14	20.78	0.23
VPC_Non_stabilizing	14.58	5.64	2.36	0.65	16.21	6.50	4.10	1.07
VPC_stabilizing	13.13	6.66	3.44	0.12	17.07	4.99	5.07	0.20

Table 4. Model checking results

	Global Time (Sec)		Sat Time (Sec)		Ratio		
Model name	Naïve	Opt	Naïve	Opt	Global	Sat	
Bcr-Abl1	69.30	9.04	26.67	0.90	7.66	29.61	sat
Bcr-Abl1	188.13	12.21	87.70	1.42	15.40	61.47	sat
Bcr-Abl1	380.24	13.12	292.21	2.01	28.96	145.02	sat
Bcr-Abl1	648.02	12.37	349.70	2.30	52.38	151.87	sat
Bcr-Abl1	1005.37	11.52	588.34	2.17	87.19	270.93	sat
Bcr-Abl2	47.04	10.97	9.94	0.72	4.28	13.76	Unsat
Bcr-Abl2	136.48	8.62	41.04	0.75	15.82	54.66	Unsat
Bcr-Abl2	285.28	11.28	112.35	0.77	25.28	144.58	Unsat
Bcr-Abl2	561.65	9.29	443.91	0.80	60.41	553.83	Unsat
Bcr-Abl2	781.64	12.03	408.55	0.87	64.96	465.55	Unsat
Bcr-Abl3	48.64	8.47	9.54	0.83	5.74	11.45	Unsat
Bcr-Abl3	133.83	9.10	38.68	1.11	14.69	34.81	Unsat
Bcr-Abl3	283.73	9.45	106.61	1.16	30.01	91.28	Unsat
Bcr-Abl3	596.50	9.50	466.01	1.18	62.78	394.48	Unsat
Bcr-Abl3	853.53	10.05	480.77	1.36	84.89	351.99	Unsat
Bcr-Abl4	75.27	9.19	44.50	0.80	8.18	55.31	sat
Bcr-Abl4	202.06	9.95	143.49	1.53	20.30	93.50	sat
Bcr-Abl4	296.02	11.35	116.24	2.54	26.07	45.74	sat
Bcr-Abl4	740.39	11.00	621.41	1.96	67.24	316.19	sat
Bcr-Abl4	975.97	10.42	823.53	1.10	93.63	747.14	sat
Bcr-AblNoFeedbacks1	42.98	6.25	7.94	0.40	6.87	19.51	Unsat
Bcr-AblNoFeedbacks1	163.33	8.18	95.43	0.77	19.95	123.90	Unsat
Bcr-AblNoFeedbacks1	302.17	6.41	122.25	0.46	47.07	260.90	Unsat
Bcr-AblNoFeedbacks1	493.28	6.41	314.24	0.45	76.92	686.28	Unsat
Bcr-AblNoFeedbacks1	809.97	6.45	680.70	0.46	125.51	1461.69	Unsat
Bcr-AblNoFeedbacks2	44.88	6.39	6.59	0.40	7.01	16.27	Unsat
Bcr-AblNoFeedbacks2	117.96	6.34	20.98	0.39	18.58	53.61	Unsat
Bcr-AblNoFeedbacks2	312.73	7.59	231.87	0.46	41.18	500.00	Unsat
Bcr-AblNoFeedbacks2	527.40	6.31	423.61	0.39	83.46	1084.74	Unsat
Bcr-AblNoFeedbacks2	751.45	6.83	362.09	0.44	109.87	806.35	Unsat
Bcr-AblNoFeedbacks3	60.99	6.95	20.45	0.64	8.77	31.64	sat
Bcr-AblNoFeedbacks3	204.66	7.06	144.58	0.61	28.97	233.95	sat
Bcr-AblNoFeedbacks3	356.33	8.81	267.48	0.49	40.42	539.32	sat
Bcr-AblNoFeedbacks3	Time out	7.06	Time out	0.42	N/A	N/A	sat
VPC_non_stabilizing1	30.14	10.83	4.83	0.69	2.78	6.93	Unsat
VPC_non_stabilizing2	17.42	9.85	3.59	1.11	1.76	3.24	sat
VPC_non_stabilizing3	52.01	11.91	26.69	1.48	4.36	17.93	Unsat
VPC_non_stabilizing4	19.53	8.31	7.08	0.60	2.34	11.77	Unsat
VPC_stabilizing1	3.75	5.11	0.31	0.07	0.73	3.99	Unsat
VPC_stabilizing2	5.53	5.32	0.86	0.11	1.04	7.41	sat

in particular the two dimensional skin model (Skin2D_5X2 from [21]) and the Leukemia model (Bcr_Abl) the new technique is an order of magnitude faster. Furthermore, when increasing the length of the path it scales a lot better than the naïve approach. When model checking is considered, the combination of the decreasing reachability sets accelerates model checking considerably. While the

naïve search increases considerably to the order of tens of minutes, the optimized search remains within the order of 10 seconds, which affords a "real-time" response to users.

6 Conclusions and Future Work

We have presented a new technique for model checking Qualitative Networks. Our technique utilizes the unique structure of Qualitative Networks to construct "decreasing reachability sets". These sets form part of a compact representation of paths in the QN and lead to significant acceleration in an implementation of bounded model checking.

Our main aim is to use these methods in order to gain new biological insights. These will be reported elsewhere; the work presented here is about the various techniques we have developed and evaluated. The tool has been made available to our biologist collaborators. We are working on adding LTL model checking as one of the supported analysis techniques in our tool BMA [2]. We find the experimental results very encouraging especially given the iterative development methodology biologists have been using when employing our tools [2]. As mentioned, our users "try out" several options and refine them according to results of simulation and verification. In this iterative process it is most important to be able to give fast answers to queries of the user. We hope that with the speed ups afforded by this new technique model checking could be incorporated into the workflow of using our tools.

We note that the encoding of target functions for highly connected variables is not efficient. Enumerating all possible options entails going over a large number of options. For example, a variable that has 8 inputs (or more) ranging over $\{0, 1, 2\}$ requires to enumerate $3^8 = 6561$ options for the transition table of one variable (not to mention adding 6561 Boolean variables per variable per state in the path – see [6]). We are currently working on alternative approaches to analyze the target function of a single variable to enable better encoding of it. We note that this is a problem also of stability analysis [8].

We intend to remove the initial states from "normal" transition systems and evaluate whether decreasing reachability sets could prove useful for other types of systems as well.

References

1. Batt, G., Ropers, D., de Jong, H., Geiselmann, J., Mateescu, R., Page, M., Schneider, D.: Validation of qualitative models of genetic regulatory networks by model checking: analysis of the nutritional stress response in escherichia coli. Bioinformatics 21(suppl. 1), i19–i28 (2005)
2. Benque, D., Bourton, S., Cockerton, C., Cook, B., Fisher, J., Ishtiaq, S., Piterman, N., Taylor, A., Vardi, M.Y.: BMA: Visual tool for modeling and analyzing biological networks. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 686–692. Springer, Heidelberg (2012)
3. Beyer, A., Thomason, P., Li, X., Scott, J., Fisher, J.: Mechanistic insights into metabolic disturbance during type-2 diabetes and obesity using qualitative networks. T. Comp. Sys. Biology 12, 146–162 (2010)

4. Bonzanni, N., Krepska, E., Feenstra, K.A., Fokkink, W., Kielmann, T., Bal, H., Heringa, J.: Executing multicellular differentiation: Quantitative predictive modelling of *C.elegans* vulval development. Bioinformatics 25(16), 2049–2056 (2009)
5. Chabrier-Rivier, N., Chiaverini, M., Danos, V., Fages, F.: Modeling and querying biomolecular interaction networks. Th. Comp. Sci. 325(1), 25–44 (2004)
6. Claessen, K., Fisher, J., Ishtiaq, S., Piterman, N., Qinsi, W.: Model-checking signal transduction networks through decreasing reachbility sets. Technical Report MSR-TR-2013-30, Microsoft Research (2013)
7. Clarke, E., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods in System Design 19(1), 7–34 (2001)
8. Cook, B., Fisher, J., Krepska, E., Piterman, N.: Proving stabilization of biological systems. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 134–149. Springer, Heidelberg (2011)
9. Dill, D., Knapp, M., Gage, P., Talcott, C., Laderoute, K., Lincoln, P.: The pathalyzer: a tool for analysis of signal transduction pathways. In: 1st Annual Recomb Satellite Workshop on Systems Biology (2005)
10. Espinosa-Sotoa, C., Padilla-Longoriab, P., Alvarez-Buyllaa, E.: A gene regulatory network model for cell-fate determination during arabidopsis thaliana flower development that is robust and recovers experimental gene expression profiles. The Plant Cell 16(11), 2923–2939 (2004)
11. Fisher, J., Henzinger, T.: Executable cell biology. Nature Biotechnology 25(11), 1239–1249 (2007)
12. Fisher, J., Piterman, N., Hajnal, A., Henzinger, T.: Predictive modeling of signaling crosstalk during *c. elegans* vulval development. PLoS Computational Biology 3(5), e92 (2007)
13. Heath, J.: The equivalence between biology and computation. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 18–25. Springer, Heidelberg (2009)
14. Heath, J., Kwiatkowska, M., Norman, G., Parker, D., Tymchyshyn, O.: Probabilistic model checking of complex biological pathways. In: Priami, C. (ed.) CMSB 2006. LNCS (LNBI), vol. 4210, pp. 32–47. Springer, Heidelberg (2006)
15. Li, F., Long, T., Lu, Y., Ouyand, Q., Tang, C.: The yeast cell-cycle network is robustly designed. Proc. Natl. Acad. Sci. U S A 101(14), 4781–4786 (2004)
16. Naldi, A., Thieffry, D., Chaouiya, C.: Decision diagrams for the representation and analysis of logical models of genetic networks. In: Calder, M., Gilmore, S. (eds.) CMSB 2007. LNCS (LNBI), vol. 4695, pp. 233–247. Springer, Heidelberg (2007)
17. Pnueli, A., Zaks, A.: On the merits of temporal testers. In: Grumberg, O., Veith, H. (eds.) 25MC Festschrift. LNCS, vol. 5000, pp. 172–195. Springer, Heidelberg (2008)
18. Ray, S., Brayton, R.: Proving stabilization using liveness to safety conversion. In: 20th International Workshop on Logic and Synthesis (2011)
19. Samal, A., Jain, S.: The regulatory network of e. coli metabolism as a boolean dynamical system exhibits both homeostasis and flexibility of response. BMC Systems Biology 2(1), 21 (2008)
20. Sanchez, L., Thieffry, D.: Segmenting the fly embryo: a logical analysis fo the *pair-rule* cross-regulatory module. Journal of Theoretical Biology 244, 517–537 (2003)
21. Schaub, M., Henzinger, T., Fisher, J.: Qualitative networks: A symbolic approach to analyze biological signaling networks. BMC Systems Biology 1(4) (2007)
22. Shmulevitch, I., Dougherty, R., Kim, S., Zhang, W.: Probabilistic boolean networks: A rule-based uncertainty model for gene regulatory networks. Bioinformatics 18(2), 261–274 (2002)
23. Thomas, R., Thieffry, D., Kaufman, M.: Dynamical behaviour of biological regulatory networks—i. biological role of feedback loops and practical use of the concept of the loop-characteristic state. Bull. of Math. Bio. 55(2), 247–276 (1995)

TTP: Tool for Tumor Progression

Johannes G. Reiter[1], Ivana Bozic[2,3], Krishnendu Chatterjee[1], and Martin A. Nowak[2,3,4]

[1] IST Austria (Institute of Science and Technology Austria), Klosterneuburg, Austria
[2] Program for Evolutionary Dynamics, Harvard University, Cambridge, USA
[3] Department of Mathematics, Harvard University, Cambridge, USA
[4] Department of Organismic and Evolutionary Biology, Harvard University, Cambridge, USA

Abstract. In this work we present a flexible tool for tumor progression, which simulates the evolutionary dynamics of cancer. Tumor progression implements a multi-type branching process where the key parameters are the fitness landscape, the mutation rate, and the average time of cell division. The fitness of a cancer cell depends on the mutations it has accumulated. The input to our tool could be any fitness landscape, mutation rate, and cell division time, and the tool produces the growth dynamics and all relevant statistics.

1 Introduction

Cancer is a genetic disease which is driven by the somatic evolution of cells [1], where driver mutations for cancer increase the reproductive rate of cells through different mechanisms, e.g. evading growth suppressors, sustaining proliferative signaling, or resisting cell death [2]. Tumors are initiated by some genetic event which increases the reproductive rate of previously normal cells. The evolution of cancer (malignant tumor) is a multi-step process where cells need to receive several mutations subsequently [3]. This phase of tumor progression is characterized by the uncontrolled growth of cells [2,4]. The requirement to accumulate multiple mutations over time explains the increased risk of cancer with age.

There are several mathematical models to explain tumor progression and the age incidence of cancer [5,6,7]. The models have also provided quantitative insights in the evolution of resistance to cancer therapy [8]. The models for tumor progression are multi-type branching processes which represent an exponentially growing heterogeneous population of cells, where the key parameters for the process are: (i) the fitness landscape of the cells (which determine the reproductive rate), (ii) the mutation rate (which determines the accumulation of driver mutations), and (iii) the average cell division time (or the generation time for new cells). The fitness landscapes allow the analysis of the effects of interdependent (driver) mutations on the evolution of cancer [9].

In this work, we present a very flexible tool (namely, TTP, tool for tumor progression) to study the dynamics of tumor progression. The input to our tool could be any fitness landscape, mutation rate, and cell division time, and the tool generates the growth dynamics and all relevant statistics (such as the expected

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 101–106, 2013.

tumor detection time, or the expected appearance time of surviving mutants, etc). Our stochastic computer simulation is an efficient simulation of a multi-type branching process under all possible fitness landscapes, driver mutation rates, and cell division times.

Our tool provides a quantitative framework to study the dynamics of tumor progression in different stages of tumor growth. Currently, the data to understand the effects of complex fitness landscapes can only be obtained from patients or animals suffering the disease. With our tool, playing with the parameters, once the real-world data is reproduced, the computer simulations can provide many simulation examples that would aid to understand these complex effects. Moreover, once the correct mathematical models for specific types of cancer are identified where the simulations match the real-world data, verification tools for probabilistic systems can be used to further analyze and understand the tumor progression process (such an approach has been followed in [10] for the verification of biological models). In this direction, results of specific fitness landscapes of our tool have already been used in a biological application paper [11]. While we present our tool for the discrete-time process (which provides a good approximation of the continuous-time process), results of our tool for the special case of a uniform fitness landscape in the continuous-time process have also been shown to have excellent agreement with the real-world data for the time to treatment failure for colorectal cancer [8]. Full version available at [12].

2 Model

Tumor progression is modeled as a discrete-time branching process (Galton-Watson process [13]). At each time step, a cell can either divide or die. The phenotype i of a cancerous cell determines its division probability b_i and is encoded as a bit string of length four (i.e. $\{0,1\}^4$). The death probability d_i follows from b_i as $d_i = 1 - b_i$. If a cell divides, one of the two daughter cells can receive an additional mutation (i.e., a bit flips from wildtype 0 to the mutated type 1) with probability u in one of the wildtype positions (e.g., cells of phenotype 1010 can receive an additional mutation only at positions two and four; cells of phenotype 1111 can not receive any additional mutations). The branching process is initiated by a single cell of phenotype $i = 0000$ (resident cell). The resident cells are wildtype at all four positions and have a strictly positive growth rate (i.e., $b_{0000} - d_{0000} > 0$).

Fitness Landscapes. Our tool provides two predefined fitness landscapes for driver mutations in tumor progression: (1) Multiplicative Fitness Landscape (MFL) and (2) Path Fitness Landscape (PFL). Additionally, the user can also define its own general fitness landscape (GFL). A fitness landscape defines the birth probability b_i for all possible phenotypes i. Following the convention of the standard modeling approaches, we let $b_{0000} = 1/2(1 + s_0)$ be the birth probability of the resident cells (i.e., cells of phenotype 0000) [9,11]. The growth coefficient s_j indicates the selective advantage provided by an additional mutation at position j in the phenotype.

Multiplicative Fitness Landscape. In the MFL a mutation at position j of the phenotype i of a cell results in a multiplication of its birth probability by $(1+s_j)$. Specifically, the birth probability b_i of a cell with phenotype i is given by

$$b_i = \frac{1}{2}(1+s_0)\prod_{j=1}^{4}(1+\widehat{s}_j);$$

where $\widehat{s}_j = 0$ if the j-th position of i is 0; otherwise $\widehat{s}_j = s_j$. Hence, each additional mutation can be weighted differently and provides a predefined effect $(1+s_1)$, $(1+s_2)$, $(1+s_3)$, or $(1+s_4)$ on the birth probability of a cell. Additional mutations can also be costly (or neutral) which can be modeled by a negative s_j (or $s_j = 0$). If $s_0 = s_1 = s_2 = s_3 = s_4$ the fitness landscape reduces to the model studied by Bozic et al. [9], which we call EMFL (Equal Multiplicative Fitness Landscape) and is also predefined in our tool.

Path Fitness Landscape. The PFL defines a certain path on which additional mutations need to occur to increase the birth probability of a cell. The predefined path can be $0000 \to 1000 \to 1100 \to 1110 \to 1111$, and again the growth coefficients s_j determine the multiplicative effect of the new mutation on the birth probability (see appendix of [12] for more details). Mutations not on this path are deleterious for the growth rate of a cell and its birth probability is set to $1/2(1-v)$. The parameter v $(0 \leq v \leq 1)$ specifies the disadvantage for cells of all phenotypes which do not belong to the given path.

General Fitness Landscapes. Our tool allows to input any fitness landscape as follows: for b_i for $i \in \{0,1\}^4$, our tool can take as input the value of b_i. In this way, any fitness landscape can be a parameter to the tool.

Density Limitation. In some situations, a tumor needs to overcome current geometric or metabolic constraints (e.g. when the tumor needs to develop blood vessels to provide enough oxygen and nutrients for further growth [14,11]). Such growth limitations are modeled by a density limit (carrying capacity) for various phenotypes. Hence, the cells of a phenotype i grow first exponentially but eventually reach a steady state around a given carrying capacity K_i. Only cells with another phenotype (additional mutation) can overcome the density limit. Logistic growth is modeled with variable growth coefficients $\widetilde{s_j} = s_j(1 - X_i/K_i)$ where X_i is the current number of cells of phenotype i in the tumor. In this model, initially $\widetilde{s_j} \approx s_j$ $(X_i \ll K_i)$, however, if X_i is on the order of K_i, $\widetilde{s_j}$ becomes approximately zero (details are given in the appendix of [12]).

3 Tool Implementation and Experimental Results

Our tool provides an efficient implementation of a very general tumor progression model. Essentially, the tool implements the above defined branching processes to simulate the dynamics of tumor growth and to obtain statistics about the

expected tumor detection time and the appearance of additional driver mutations during different stages of disease progression. TTP can be downloaded from here: `http://pub.ist.ac.at/ttp`.

For an efficient processing of the discrete-time branching process, the stochastic simulation samples from a multinomial distribution for each phenotype at each time step [9,11]. The sample returns the number of cells which divided with and without mutation and the number of cells which died in the current generation (see the appendix of [12] for details). From the samples for each phenotype the program calculates the phenotype distribution in the next generation. Hence, the program needs to store only the number of cells of each phenotype during the simulation. This efficient implementation of the branching process allows the tool to simulate many patients within a second and to obtain very good statistical results in a reasonable time frame.

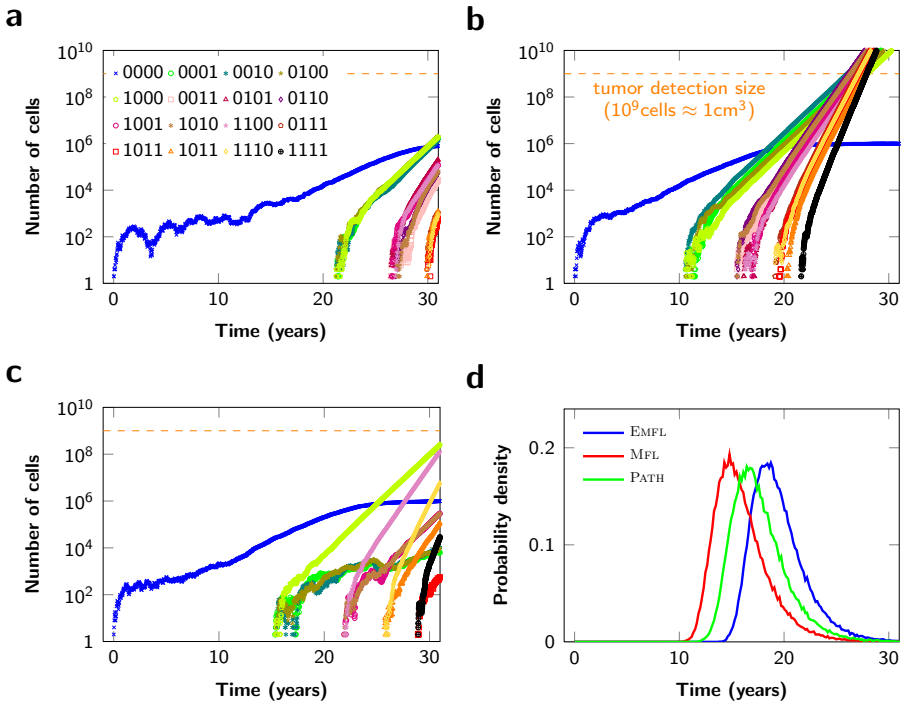


Fig. 1. Experimental results illustrating the variability of tumor progression. In panels a and b we show examples for two particular simulation runs where the cells grow according to the EMFL and resident cells (blue) are constrained by a carrying capacity of $K_{0000} = 10^6$. In panel c the cells grow according to the PFL. In panel d we show statistical results for the probability density of tumor detection when cells grow according to different fitness landscapes. Parameter values: growth coefficients $s_0 = 0.004$, $s_1 = 0.006$, $s_2 = 0.008$, $s_3 = 0.01$, $s_4 = 0.012$, and $v = 0.01$, mutation rate $u = 10^{-6}$, cell division time $T = 3$ days, tumor detection size 10^9 cells.

Modes. The tool can run in the following two modes: individual or statistics. In the *individual mode* the tool produces the growth dynamics of one tumor in a patient (see panels a, b, and c in Fig. 1). Furthermore, both the growth dynamics and the phenotype distribution of the tumor are depicted graphically. In the *statistics mode* the tool produces the probability distribution for the detection time of the tumor (see panel d in Fig. 1) both graphically and quantitatively. Additionally, the tool calculates for all phenotypes the appearance times of the first surviving lineage, the existence probability, and the average number of cells at detection time.

Features. TTP provides an intuitive graphical user interface to enter the parameters of the model and shows plots of the dynamics during tumor progression, the phenotype distribution, or the probability density of tumor detection. These plots can also be saved as files in various image formats. Furthermore, the tool can create data files (tab-separated values) of the tumor growth history and the probability distribution of tumor detection for any set of input parameters (details on the format are given in the appendix).

Input Parameters. In both modes, the tool takes the following input parameters: (i) growth coefficients s_0, s_1, s_2, s_3, and s_4 (and v in the case of PFL), (ii) mutation rate u, (iii) cell generation time T, (iv) fitness landscape (MFL, PFL, EMFL, or GFL with the birth probability for each phenotype), and optional (v) density limits for some phenotypes. In the individual mode, additionally, the user needs to provide the number of generations which have to be simulated. In the statistics mode, the additional parameters are: the tumor detection size and the number of patients (tumors which survive the initial stochastic fluctuations) which have to be simulated.

Experimental Results. In panels a, b, and c of Fig. 1 we show examples of the growth dynamics during tumor progression. Although we used exactly the same parameters in panels a and b, we observe that the time from tumor initiation until detection can be very different. In panel d we show the probability density of tumor detection under various fitness landscapes. Further experimental results are given in the appendix.

Case Studies. Several results of these models have shown excellent agreement with different aspects of real-world data. In [9], results for the expected tumor size at detection time using a EMFL fit the reported polyp sizes of the patients very well. Similarly, using a continuous-time branching process and a uniform fitness landscape, results for the expected time to the relapse of a tumor after start of treatment agree thoroughly with the observed times in patients [8].

Future Work. In some ongoing work, we also investigate mathematical models for tumor dynamics occurring during cancer treatment modeled by a continuous-time branching process. Thus an interesting extension of our tool would be to model treatment as well. Another interesting direction is to model the seeding of metastasis during tumor progression and hence simulate a "full" patient rather than the primary tumor alone. Once faithful models of the evolution of cancer have been identified, verification tools such as PRISM [15] and theoretical results such as [16] might contribute to the understanding of these processes.

Acknowledgments. This work is supported by the ERC Start grant (279307: Graph Games), the FWF NFN Grant (No S11407-N23, Rise), the FWF Grant (No P 23499-N23), a Microsoft Faculty Fellow Award, the Foundational Questions in Evolutionary Biology initiative of the John Templeton Foundation, and the NSF/NIH joint program in mathematical biology (NIH grant R01GM078986).

References

1. Vogelstein, B., Kinzler, K.W.: Cancer genes and the pathways they control. Nature Medicine 10(8), 789–799 (2004)
2. Hanahan, D., Weinberg, R.A.: Hallmarks of cancer: the next generation. Cell 144(5), 646–674 (2011)
3. Jones, S., Chen, W.D., Parmigiani, G., Diehl, F., Beerenwinkel, N., Antal, T., Traulsen, A., Nowak, M.A., Siegel, C., Velculescu, V.E., Kinzler, K.W., Vogelstein, B., Willis, J., Markowitz, S.D.: Comparative lesion sequencing provides insights into tumor evolution. PNAS 105(11), 4283–4288 (2008)
4. Nowak, M.A.: Evolutionary Dynamics: Exploring the equations of life. The Belknap Press of Harvard University Press, Cambridge (2006)
5. Komarova, N.L., Sengupta, A., Nowak, M.A.: Mutation-selection networks of cancer initiation: tumor suppressor genes and chromosomal instability. Journal of Theoretical Biology 223(4), 433–450 (2003)
6. Iwasa, Y., Michor, F., Nowak, M.A.: Stochastic tunnels in evolutionary dynamics. Genetics 166(3), 1571–1579 (2004)
7. Nowak, M.A., Michor, F., Komarova, N.L., Iwasa, Y.: Evolutionary dynamics of tumor suppressor gene inactivation. PNAS 101(29), 10635–10638 (2004)
8. Diaz, L.A., Williams, R.T., Wu, J., Kinde, I., Hecht, J.R., Berlin, J., Allen, B., Bozic, I., Reiter, J.G., Nowak, M.A., Kinzler, K.W., Oliner, K.S., Vogelstein, B.: The molecular evolution of acquired resistance to targeted EGFR blockade in colorectal cancers. Nature 486(7404), 537–540 (2012)
9. Bozic, I., Antal, T., Ohtsuki, H., Carter, H., Kim, D., Chen, S., Karchin, R., Kinzler, K.W., Vogelstein, B., Nowak, M.A.: Accumulation of driver and passenger mutations during tumor progression. PNAS 107(43), 18545–18550 (2010)
10. Sadot, A., Fisher, J., Barak, D., Admanit, Y., Stern, M., Hubbard, E., Harel, D.: Toward verified biological models. IEEE/ACM Transactions on Computational Biology and Bioinformatics 5(2), 223–234 (2008)
11. Reiter, J.G., Bozic, I., Allen, B., Chatterjee, K., Nowak, M.A.: The effect of one additional driver mutation on tumor progression. Evolutionary Applications 6(1), 34–45 (2013)
12. Reiter, J.G., Bozic, I., Chatterjee, K., Nowak, M.A.: TTP: Tool for Tumor Progression. CoRR abs/1303.5251 (2013), `http://arxiv.org/abs/1303.5251`
13. Haccou, P., Jagers, P., Vatutin, V.A.: Branching Processes: Variation, Growth, and Extinction of Populations. Cambridge University Press (2005)
14. Kerbel, R.S.: Tumor angiogenesis: past, present and the near future. Carcinogenesis 21(3), 505–515 (2000)
15. Hinton, A., Kwiatkowska, M.Z., Norman, G., Parker, D.: Prism: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
16. Etessami, K., Stewart, A., Yannakakis, M.: Polynomial time algorithms for multitype branching processesand stochastic context-free grammars. In: STOC, pp. 579–588 (2012)

Exploring Parameter Space of Stochastic Biochemical Systems Using Quantitative Model Checking*

Luboš Brim, Milan Češka, Sven Dražan, and David Šafránek

Systems Biology Laboratory at Faculty of Informatics, Masaryk University,
Botanická 68a, 602 00 Brno, Czech Republic
{brim,xceska,xdrazan,xsafran1}@fi.muni.cz

Abstract. We propose an automated method for exploring kinetic parameters of stochastic biochemical systems. The main question addressed is how the validity of an *a priori* given hypothesis expressed as a temporal logic property depends on kinetic parameters. Our aim is to compute a landscape function that, for each parameter point from the inspected parameter space, returns the quantitative model checking result for the respective continuous time Markov chain. Since the parameter space is in principle dense, it is infeasible to compute the landscape function directly. Hence, we design an effective method that iteratively approximates the lower and upper bounds of the landscape function with respect to a given accuracy. To this end, we modify the standard uniformization technique and introduce an iterative parameter space decomposition. We also demonstrate our approach on two biologically motivated case studies.

1 Introduction

The importance of stochasticity in biochemical processes having low numbers of molecules has resulted in the development of stochastic models [12]. Stochastic biochemical processes can be faithfully modeled as continuous time Markov chains (CTMCs) [9]. Knowledge of stochastic rate constants (model parameters) is important for the analysis of system dynamics. Moreover, knowledge about how change in parameters influences system dynamics (parameter exploration) is of great importance in tuning the stochastic model. Prior knowledge of kinetic parameters is usually limited. The model identification routine thus typically includes parameter estimation based on experimental data. While parameter exploration and estimation is well-established for deterministic models, it has not yet been adequately addressed and sufficiently developed for stochastic models. The purpose of this work is to develop practical and effective methods for exact exploration of model parameters in stochastic biochemical models.

The main question addressed is how the validity of an *a priori* given hypothesis expressed as a temporal property depends on model parameters. Parameter estimation

* This work has been supported by the Czech Science Foundation grant No. GAP202/11/0312. M. Češka has been supported by Ministry of Education, Youth, and Sport project No. CZ.1.07/2.3.00/30.0009 – Employment of Newly Graduated Doctors of Science for Scientific Excellence. D. Šafránek has been supported by EC OP project No. CZ.1.07/2.3.00/20.0256.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 107–123, 2013.

gives a single point in the parameter space where the values of model parameters maximize the agreement of model behaviour with experimental data. On the contrary, we often do not want to have a single objective but rather explore the property over the entire parameter space. Our main goal is to compute a *landscape function* that for each parameter point from the inspected parameter space returns the quantitative model checking result for the respective CTMC determined by the parameter point and the given property. Since the inspected parameter space is in principle dense the set of parametrized CTMCs to be explored is infinite. It is thus not possible to compute the model checking result for each CTMC individually.

As a temporal logic we use the *bounded time fragment of Continuous Stochastic Logic (CSL)* [2] further extended with rewards [19]. For most cases of biochemical stochastic systems the bounded time restriction is adequate since a typical behaviour is recognizable in finite time intervals.

In this paper we consider the *parameter exploration problem* for stochastic biochemical systems in terms of a landscape function that returns for each parameter point the probability or the expected reward of the inspected CSL formula. We propose a method, called *min-max approximation*, that computes the *lower* and *upper* approximations of the landscape function. To compute the approximation for an arbitrary nested CSL formula, we introduce the largest and smallest set of states satisfying the formula and show how to compute such sets effectively using a new method called *parametrized uniformization*. To compute the landscape function approximation with given accuracy we employ *iterative parameter space decomposition* that divides the parameter space into subspaces and allows to compute the proposed approximation independently for each subspace. This decomposition refines the approximation and enables to reach the required accuracy bound. We demonstrate our approach on two biologically motivated case studies. In the first one, we demonstrate that parametrized uniformization allows to approximate the transient probabilities of Schloegel's model [24] for the inspected parameter space. In the second case, our method is applied to parameter exploration of bi-stability in mammalian cell cycle gene regulatory control [25]. Several techniques have been employed [26,10] to analyze models of this kind, especially, it has been shown that asymptotic solutions may disagree with the exact solution imposing thus a challenge for more accurate computational techniques. Since in low molecular numbers stochasticity can produce behaviour that significantly differs from asymptotic and deterministic dynamics, the parameter exploration method reflecting this phenomenon is very important for computational systems biology.

In contrast to methods mentioned in the related work section, the accuracy of results can be fully controlled and adjusted by the user. Similarly to these methods our method is computationally intensive. However, it can be easily parallelized since the computation for each subspace is independent. Moreover, it can be also combined with fast adaptive uniformization [9] and sliding window abstraction [16].

Related Work. To the best of our knowledge there is no other work on stochastic models employing CSL model checking to systematic parameter exploration. The closest work is [22] where a CTMC is explored with respect to a property formalized as a deterministic timed automaton (DTA). It extends [1] to parameter estimation with respect to acceptance of the DTA. Approaches to parameter estimation [23,1,7] rely on

approximating the maximum likelihood. Their advantage is the possibility to analyse infinite state spaces [1] (employing dynamic state space truncation with numerically computed likelihood) or even models with no prior knowledge of parameter ranges [7] (using Monte-Carlo optimization for computing the likelihood). All these methods are not suitable for computing the landscape function since they focus on optimizing a *single objective* and not on global *exploration* of the entire requested parameter space.

Approaches based on Markov Chain Monte-Carlo sampling and Bayesian inference [13,17,18] can be extended to sample-based approximation of the landscape function, but at the price of undesired inaccuracy and high computational demands [6,4]. Compared to these methods, our method provides an exact result without neglecting any singularities caused by possible discontinuities in the landscape function.

In [15], for a given parametrized discrete time Markov chain (DTMC) the problem of synthesis for a probabilistic temporal logic is considered. The problem is reduced to constructing a regular expression representing the property validity while addressing the problem of expression explosion. Construction of the expression and also the proposed reduction techniques rely on the discrete nature of DTMC. These techniques cannot be successfully applied to CTMC since the complexity of the expression is given by maximal number of events that can occur within the inspected time horizon. In a typical biochemical system where time scales of individual reactions differ in several orders the number of reactions that can occur is enormous.

Barbuti et al. [5] treat stochastic biochemical models with parameter uncertainty in terms of interval discrete time Markov chains. They reduce quantitative reachability analysis of uncertain models to reachability analysis of a Markov Decision Process. However, no analogy to landscape function and automatized parameter decomposition is considered. Moreover, our method deals with continuous time semantics.

2 Background

Stochastic Biochemical Systems. A finite state stochastic biochemical system $\mathcal{S}$ is defined by a set of N *chemical species* in a well stirred volume with fixed size and fixed temperature participating in M *chemical reactions*. The number X_i of molecules of each species S_i has a specific bound and each reaction is of the form $u_1S_1 + \ldots + u_NS_N \longrightarrow v_1S_1 + \ldots + v_NS_N$ where $u_i, v_i \in \mathbb{N}_0$ represent *stoichiometric coefficients*.

A *state* of a system in time t is the vector $\mathbf{X}(t) = (X_1(t), X_2(t), \ldots, X_N(t))$. When a single reaction with index $r \in \{1, \ldots, M\}$ with vectors of stoichiometric coefficients U_r and V_r occurs the state changes from $\mathbf{X}$ to $\mathbf{X}' = \mathbf{X} - U_r + V_r$, which we denote as $\mathbf{X} \xrightarrow{r} \mathbf{X}'$. For such reaction to happen in a state $\mathbf{X}$ all reactants have to be in sufficient numbers and the state $\mathbf{X}'$ must reflect all species bounds. The *reachable state space* of $\mathcal{S}$, denoted as $\mathbb{S}$, is the set of all states reachable by a finite sequence of reactions from *an initial state* $\mathbf{X}_0$. For each state $\mathbf{X}_i$ we denote $\mathsf{pred}(\mathbf{X}_i) = \{(j,r) \mid \mathbf{X}_j \xrightarrow{r} \mathbf{X}_i\}$ and $\mathsf{succ}(\mathbf{X}_i) = \{(j,r) \mid \mathbf{X}_i \xrightarrow{r} \mathbf{X}_j\}$ the sets of all predecessors and successors, respectively, together with indices of corresponding reactions. The set of indices of all reactions changing the state $\mathbf{X}_i$ to the state $\mathbf{X}_j$ is denoted as $\mathsf{reac}(\mathbf{X}_i, \mathbf{X}_j) = \{r \mid \mathbf{X}_i \xrightarrow{r} \mathbf{X}_j\}$. Henceforward the reactions will be referred directly by their indices.

According to Gillespie [12] the behaviour of a stochastic system $\mathcal{S}$ can be described by the continuous time Markov chain (CTMC) $\mathcal{C} = (\mathbb{S}, \mathbf{X}_0, \mathbf{R})$ where the transition matrix $\mathbf{R}(i,j)$ gives the probability of a transition from $\mathbf{X}_i$ to $\mathbf{X}_j$. Formally, $\mathbf{R}(i,j) = \sum_{r \in \mathsf{reac}(\mathbf{X}_i, \mathbf{X}_j)} k_r \cdot C_{r,i}$ such that k_r is *a stochastic rate constant* of the reaction r and $C_{r,i} \stackrel{def}{=} \prod_{l=1}^{N} \binom{\mathbf{X}_{i,l}}{u_l}$ corresponds to the population dependent term of the *propensity function* where $\mathbf{X}_{i,l}$ is lth component of the state $\mathbf{X}_i$ and u_l is the stoichiometric coefficient of the reactant S_l in reaction r.

Parameter Space. Let each stochastic rate constant k_i have a value interval $[k_i^{\perp}, k_i^{\top}]$ with minimal and maximal bounds expressing *uncertainty range* of its value. *A parameter space* $\mathbf{P}$ induced by a set of stochastic rate constants k_i is defined as the Cartesian product of the individual value intervals $\mathbf{P} = \prod_{i=1}^{M} [k_i^{\perp}, k_i^{\top}]$. A single *parameter point* $p \in \mathbf{P}$ is an M-tuple holding a single value of each rate constant $p = (k_{1_p}, \ldots, k_{M_p})$. We consider only independent parameters, however, if correlated parameters can be expressed as linear functions then our method can be still applied.

A stochastic system $\mathcal{S}_p$ with its stochastic rate constants set to the point $p \in \mathbf{P}$ is represented by a CTMC $\mathcal{C}_p = (\mathbb{S}, \mathbf{X}_0, \mathbf{R}_p)$ where transition matrix $\mathbf{R}_p$ is defined as $\mathbf{R}_p(i,j) = \sum_{r \in \mathsf{reac}(\mathbf{X}_i, \mathbf{X}_j)} k_{r_p} \cdot C_{r,i}$. A *set of parametrized* CTMCs induced by the parameter space $\mathbf{P}$ is defined as $\mathbf{C} = \{\mathcal{C}_p \mid p \in \mathbf{P}\}$. Henceforward, the states $\mathbf{X}_i \in \mathbb{S}$ will be denoted as s_i.

Uniformization. Uniformization is a standard technique that for a given CTMC $\mathcal{C} = (\mathbb{S}, s_0, \mathbf{R})$ computes the transient probability in time t. For an initial state s_0 it returns a vector $\pi^{\mathcal{C},s_0,t}$ such that $\pi^{\mathcal{C},s_0,t}(s') = Pr_{s_0}\{\omega \in Path^{\mathcal{C}}(s_0) \mid \omega@t = s'\}$ for all states $s' \in \mathbb{S}$, where Pr_{s_0} is a unique probability measure on all paths ω starting in state s_0 (denoted as $Path^{\mathcal{C}}(s_0)$) defined, e.g., in [21] and $\omega@t$ is the state on path ω occupied at time t.

The transient probability in time t is obtained as a sum of expressions giving the state distributions after i discrete reaction steps weighted by the ith Poisson probability $\gamma_{i,q\cdot t} = e^{-q\cdot t} \cdot \frac{(q\cdot t)^i}{i!}$, the probability of i such steps occurring up to t, given the delay is exponentially distributed with *rate* q. Formally, $\pi^{\mathcal{C},s_0,t} = \sum_{i=0}^{\infty} \gamma_{i,q\cdot t} \cdot \pi^{\mathcal{C},s_0,0} \cdot (\mathbf{Q}^{\mathsf{unif}(\mathcal{C})})^i \approx \sum_{i=L_\varepsilon}^{R_\varepsilon} \gamma_{i,q\cdot t} \cdot \pi^{\mathcal{C},s_0,0} \cdot (\mathbf{Q}^{\mathsf{unif}(\mathcal{C})})^i$ where $\mathbf{Q}^{\mathsf{unif}(\mathcal{C})}$ is an *uniformized infinitesimal generator matrix* defined as follows: $\mathbf{Q}^{\mathsf{unif}(\mathcal{C})}(s,s') = \frac{\mathbf{R}(s,s')}{q}$, if $s \neq s'$, and $1 - \sum_{s'' \neq s} \frac{\mathbf{R}(s,s'')}{q}$, otherwise, where $q \geq max\{E^{\mathcal{C}}(s) \mid s \in \mathbb{S}\}$ such that $E^{\mathcal{C}}(s) = \sum_{s' \in \mathbb{S}} \mathbf{R}(s,s')$ is *an exit rate* of the state s in CTMC $\mathcal{C}$. Although the sum is in general infinite, for a given precision ε the upper and lower bounds $L_\varepsilon, R_\varepsilon$ can be estimated by using techniques such as of Fox and Glynn [11] which also allow for efficient computation of Poisson probabilities $\gamma_{i,q\cdot t}$. In order to make the computation feasible the matrix-matrix multiplication is reduced to a vector-matrix multiplication by pre-multiplying, i.e., $\pi^{\mathcal{C},s_0,0} \cdot (\mathbf{Q}^{\mathsf{unif}(\mathcal{C})})^i = (\pi^{\mathcal{C},s_0,0} \cdot (\mathbf{Q}^{\mathsf{unif}(\mathcal{C})})^{i-1}) \cdot \mathbf{Q}^{\mathsf{unif}(\mathcal{C})}$.

Property Specification. We consider the *bounded time fragment of CSL with rewards*, see [19] for definition of CSL with rewards. The fragment syntax is defined as follows. A state formula Φ is given as $\Phi ::= \mathsf{true} \mid a \mid \neg\Phi \mid \Phi \wedge \Phi \mid \mathsf{P}_{\sim p}[\phi] \mid \mathsf{R}_{\sim r}[\mathsf{C}^{\leq t}] \mid \mathsf{R}_{\sim r}[\mathsf{I}^{=t}]$

where ϕ is a path formula given as $\phi ::= \mathsf{X}\,\Phi \mid \Phi\ \mathsf{U}^I\ \Phi$, a is an atomic proposition, $\sim \in \{<, \leq, \geq, >\}$, $p \in [0,1]$ is a probability, $r \in \mathbb{R}_{\geq 0}$ is an expected reward and $I = [a,b]$ is a bounded time interval such that $a,b \in \mathbb{R}_{\geq 0} \wedge a \leq b$. Operators G and F can be derived in the standard way. In order to specify the reward properties, CTMCs are enhanced with reward (cost) structures. Two types of reward structure are used. *A state reward* $\rho(s)$ defines the rate with which a reward is acquired in state $s \in \mathbb{S}$. A reward of $t \cdot \rho(s)$ is acquired if a CTMC remains in state s for t time units. *A transition reward* $\iota(s_i, s_j)$ defines the reward acquired each time the transition (s_i, s_j) occurs.

Let $\mathcal{C} = (\mathbb{S}, s_0, \mathbf{R}, L)$ be a labelled CTMC such that L is a labelling function which assigns to each state $s \in \mathbb{S}$ the set $L(s)$ of atomic propositions that are valid in state s. A state s satisfies $\mathsf{P}_{\sim p}[\phi]$ (denoted as $s \vDash \mathsf{P}_{\sim p}[\phi]$) iff $Prob^{\mathcal{C}}(s,\phi) \stackrel{def}{=} Pr_s\{\omega \in Path^{\mathcal{C}}(s) \mid \omega \vDash \phi\}$ satisfies $\sim p$. A path ω satisfies $\mathsf{X}\,\Phi$ iff $\omega(1) \vDash \Phi$ where $\omega(1)$ is the second state on ω. A path ω satisfies $\Phi\ \mathsf{U}^I\ \Psi$ iff $\exists t \in I.(\omega@t \vDash \Psi \wedge \forall t' \in [0,t).(\omega@t' \vDash \Phi))$.

Intuitively, a state $s \vDash \mathsf{R}_{\sim p}[\mathsf{C}^{\leq t}]$ iff the sum of expected rewards over $Path^{\mathcal{C}}(s)$ *cumulated* until t time units (denoted as $Exp^{\mathcal{C}}(s, \mathsf{X}_{\mathsf{C}^{\leq t}})$) satisfies $\sim p$. Similarly, a state $s \vDash \mathsf{R}_{\sim p}[\mathsf{I}^{=t}]$ iff the sum of expected rewards over all paths $\omega \in Path^{\mathcal{C}}(s)$ at time t (denoted as $Exp^{\mathcal{C}}(s, \mathsf{X}_{\mathsf{I}^{=t}})$) satisfies $\sim p$. A set $Sat_C(\Phi) = \{s \in \mathbb{S} \mid s \vDash \Phi\}$ denotes the set of states that satisfy Φ.

The formal semantics of this fragment is defined similarly as the semantics of full CSL and thus we refer the readers to original papers. In the following text all references to CSL address this fragment. Model checking of CSL can be easily reduce to the computation of transient probability, see [3,21] for more details.

3 Parameter Exploration

In this paper we propose an effective method for systematic and fully automatic parameter exploration of a given stochastic system with respect to a specified temporal property and a parameter space. Let $\mathbf{C}$ be a set of parametrized CTMCs describing the dynamics of the stochastic system $\mathcal{S}$ induced by the inspected parameter space $\mathbf{P}$ and a CSL formula Φ expressing the required behaviour. The problem of *parameter exploration* is as follows: for each state $s \in \mathbb{S}$ compute the *landscape function* $\lambda_s^{\Phi,\mathbf{P}} : \mathbf{P} \to \mathbb{R}_{\geq 0}$ that for each parameter point $p \in \mathbf{P}$ returns the numerical value of the probability or the expected reward for the formula Φ. It means that we consider "quantitative" formulae in the form $\Phi ::= \mathsf{P}_{=?}[\phi] \mid \mathsf{R}_{=?}[\mathsf{C}^{\leq t}] \mid \mathsf{R}_{=?}[\mathsf{I}^{=t}]$, i.e., the topmost operator of the formula Φ returns a quantitative result, as used, e.g., in PRISM [20]. Note that the formula Φ can contain nested probabilistic and reward operators whose evaluations define discrete sets of states further used in the computation of the resulting numerical value. Therefore, the corresponding landscape function is not in general continuous but only piecewise continuous. Also note that the landscape function is inherently bounded.

To solve the parameter exploration problem we extend *global quantitative model checking* techniques enabling to compute for all states of a CTMC the numerical value of the probability or the expected reward for formula Φ. The most crucial part of the problem is given by the fact that the parameter space $\mathbf{P}$ is continuous and thus the set $\mathbf{C}$ is infinite. Therefore, it is not possible to employ the global quantitative model checking techniques for each CTMC $\mathcal{C}_p \in \mathbf{C}$ individually.

Our approach to this problem is based on a new technique which we call *min-max approximation*. The key idea is to approximate the landscape function $\lambda_s^{\Phi,\mathbf{P}}$ using a lower bound $\overline{min}_s^{\Phi,\mathbf{P}} = min\{\lambda_s^{\Phi,\mathbf{P}}(p) \mid p \in \mathbf{P}\}$ and an upper bound $\overline{max}_s^{\Phi,\mathbf{P}} = max\{\lambda_s^{\Phi,\mathbf{P}}(p) \mid p \in \mathbf{P}\}$. Since the computation of the exact bounds is computationally infeasible, we further approximate these bounds, i.e., we compute approximations $min_s^{\Phi,\mathbf{P}}$ and $max_s^{\Phi,\mathbf{P}}$ such that $min_s^{\Phi,\mathbf{P}} \leq \overline{min}_s^{\Phi,\mathbf{P}}$ and $max_s^{\Phi,\mathbf{P}} \geq \overline{max}_s^{\Phi,\mathbf{P}}$. Although the proposed min-max approximation provides the lower and upper bounds of the landscape function, it introduces an *inaccuracy* with respect to parameter exploration, i.e., such approximation can be insufficient for the inspected parameter space $\mathbf{P}$ and the given formula Φ. Formally, the inaccuracy for a state s is given as the difference $max_s^{\Phi,\mathbf{P}} - min_s^{\Phi,\mathbf{P}}$.

A significant advantage of the min-max approximation is that it allows us to iteratively decrease the inaccuracy to a required bound. The key idea is based on *iterative parameter space decomposition* where the parameter space $\mathbf{P}$ is divided into subspaces that are processed independently. The result of such computation is an approximation of the lower bound $min_s^{\Phi,\mathbf{P}_i}$ and the upper bound $max_s^{\Phi,\mathbf{P}_i}$ for each subspace $\mathbf{P}_i$. Such decomposition provides more precise approximation of the landscape function $\lambda_s^{\Phi,\mathbf{P}}$ and enables to reach the required accuracy bound.

In order to effectively compute the min-max approximation for the given formula we design a new method called *parametrized uniformization* allowing to efficiently approximate the transient probabilities for the set $\mathbf{C}$ of parametrized CTMCs. The key idea is to modify *standard uniformization* [14] in such a way that an approximation of the minimal and maximal transient probability with respect to the set $\mathbf{C}$ can be computed. Moreover, the proposed modification preserves the asymptotic time complexity of standard uniformization. Following the model checking method for non-parametrized CTMC presented in [3,21], the result of parametrized uniformization is further used to obtain the min-max approximation of the landscape function $\lambda_s^{\Phi,\mathbf{P}}$.

We are aware that the landscape function could be computed by using standard uniformization to obtain precise values in grid points which could be afterwards interpolated linearly or polynomially. Using adaptive grid refinement such an approach could also provide an arbitrary degree of precision with computation complexity of the same asymptotic class as our method. However, the obtained result would be a general approximation not providing the strict minimal and maximal upper bounds. On the contrary, our min-max approximation guarantees upper and lower estimates without neglecting any singularities caused by possible discontinuities in the landscape function that we consider to be an important feature.

4 Min-Max Approximation

To effectively compute the proposed min-max approximation for an arbitrary nested CSL formula we introduce *the largest and smallest set of states satisfying property* Φ. Let $\mathbf{C}$ be a set of labelled parametrized CTMCs over the parameter space $\mathbf{P}$ such that $\mathbf{C} = \{\mathcal{C}_p \mid p \in \mathbf{P}\}$ where each $\mathcal{C}_p = (\mathbb{S}, s_0, \mathbf{R}_p, L)$. The maximal set of states satisfying Φ, denoted by $\overline{Sat}_{\mathbf{C}}^{\top}(\Phi)$, is defined as $\overline{Sat}_{\mathbf{C}}^{\top}(\Phi) \stackrel{def}{=} \bigcup_{\mathcal{C}_p \in \mathbf{C}} Sat_{\mathcal{C}_p}(\Phi)$. The minimal set of states satisfying Φ, denoted by $\overline{Sat}_{\mathbf{C}}^{\perp}(\Phi)$, is defined as $\overline{Sat}_{\mathbf{C}}^{\perp}(\phi) \stackrel{def}{=} \bigcap_{\mathcal{C}_p \in \mathbf{C}} Sat_{\mathcal{C}_p}(\Phi)$.

Since the set $\mathbf{C}$ is not finite, this definition is not constructive and does not allow to obtain the sets $\overline{Sat}^{\top}_{\mathbf{C}}(\Phi)$ and $\overline{Sat}^{\bot}_{\mathbf{C}}(\Phi)$. Therefore, we define satisfaction relations $\models_{\top}$ and $\models^{\bot}$ that give an alternative characterization of these sets and allow us to effectively compute their approximations.

For any state $s \in \mathbb{S}$ relations $s \models_{\top} \Phi$ and $s \models_{\bot} \Phi$ are defined inductively by:

$$
\begin{array}{ll|ll}
s \models_{\top} \mathsf{true} \wedge s \models_{\bot} \mathsf{true}, \text{ for all } s \in \mathbb{S} & & s \models_{\top} a \Leftrightarrow s \models_{\bot} a \Leftrightarrow a \in L(s) & \\
s \models_{\top} \neg\Phi & \Leftrightarrow s \not\models_{\bot} \Phi & s \models_{\bot} \neg\Phi & \Leftrightarrow s \not\models_{\top} \Phi \\
s \models_{\top} \Phi \wedge \Psi & \Leftrightarrow s \models_{\top} \Phi \wedge s \models_{\top} \Psi & s \models_{\bot} \Phi \wedge \Psi & \Leftrightarrow s \models_{\bot} \Phi \wedge s \models_{\bot} \Psi \\
s \models_{\top} \mathsf{P}_{\leq p}[\phi] & \Leftrightarrow \overline{Prob}^{\mathbf{C}}_{\bot}(s,\phi) \leq p & s \models_{\bot} \mathsf{P}_{\leq p}[\phi] & \Leftrightarrow \overline{Prob}^{\mathbf{C}}_{\top}(s,\phi) \leq p \\
s \models_{\top} \mathsf{P}_{\geq p}[\phi] & \Leftrightarrow \overline{Prob}^{\mathbf{C}}_{\top}(s,\phi) \geq p & s \models_{\bot} \mathsf{P}_{\geq p}[\phi] & \Leftrightarrow \overline{Prob}^{\mathbf{C}}_{\bot}(s,\phi) \geq p \\
s \models_{\top} \mathsf{R}_{\leq p}[\mathsf{I}^{=t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}_{\mathsf{I}^{=t}}) \leq p & s \models_{\bot} \mathsf{R}_{\leq p}[\mathsf{I}^{=t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}_{\mathsf{I}^{=t}}) \leq p \\
s \models_{\top} \mathsf{R}_{\geq p}[\mathsf{I}^{=t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}_{\mathsf{I}^{=t}}) \geq p & s \models_{\bot} \mathsf{R}_{\geq p}[\mathsf{I}^{=t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}_{\mathsf{I}^{=t}}) \geq p \\
s \models_{\top} \mathsf{R}_{\leq p}[\mathsf{C}^{\leq t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}_{\mathsf{C}^{\leq t}}) \leq p & s \models_{\bot} \mathsf{R}_{\leq p}[\mathsf{C}^{\leq t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}_{\mathsf{C}^{\leq t}}) \leq p \\
s \models_{\top} \mathsf{R}_{\geq p}[\mathsf{C}^{\leq t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}_{\mathsf{C}^{\leq t}}) \geq p & s \models_{\bot} \mathsf{R}_{\geq p}[\mathsf{C}^{\leq t}] & \Leftrightarrow \overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}_{\mathsf{C}^{\leq t}}) \geq p
\end{array}
$$

where

$$
\begin{aligned}
\overline{Prob}^{\mathbf{C}}_{\top}(s,\phi) &\stackrel{def}{=} max\{Prob^{\mathcal{C}_p}(s,\phi) \mid \mathcal{C}_p \in \mathbf{C}\} \\
\overline{Prob}^{\mathbf{C}}_{\bot}(s,\phi) &\stackrel{def}{=} min\{Prob^{\mathcal{C}_p}(s,\phi) \mid \mathcal{C}_p \in \mathbf{C}\} \\
\overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}) &\stackrel{def}{=} max\{Exp^{\mathcal{C}_p}(s,\mathsf{X}) \mid \mathcal{C}_p \in \mathbf{C}\} \text{ for } \mathsf{X} \in \{\mathsf{X}_{\mathsf{I}^{=t}}, \mathsf{X}_{\mathsf{C}^{\leq t}}\} \\
\overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}) &\stackrel{def}{=} min\{Exp^{\mathcal{C}_p}(s,\mathsf{X}) \mid \mathcal{C}_p \in \mathbf{C}\} \text{ for } \mathsf{X} \in \{\mathsf{X}_{\mathsf{I}^{=t}}, \mathsf{X}_{\mathsf{C}^{\leq t}}\}
\end{aligned}
$$

By structural induction it can be proved that $\forall s \in \mathbb{S} : s \in \overline{Sat}^{\top}_{\mathbf{C}}(\Phi) \Rightarrow s \models_{\top} \Phi$ and $s \models_{\bot} \Phi \Rightarrow s \in \overline{Sat}^{\bot}_{\mathbf{C}}(\Phi)$. This characterization allows us to define an approximation $Sat^{\top}_{\mathbf{C}}(\Phi)$ and $Sat^{\bot}_{\mathbf{C}}(\Phi)$ in the following way. For all $s \in \mathbb{S} : s \in Sat^{\top}_{\mathbf{C}}(\Phi) \stackrel{def}{\Leftrightarrow} s \models^{*}_{\top} \Phi$ and $s \in Sat^{\bot}_{\mathbf{C}}(\Phi) \stackrel{def}{\Leftrightarrow} s \models^{*}_{\bot} \Phi$ where the definition of $\models^{*}_{\top}$ and $\models^{*}_{\bot}$ differs from the definition of $\models_{\top}$ and $\models_{\bot}$ such that the exact values $\overline{Prob}^{\mathbf{C}}_{\top}(s,\phi)$, $\overline{Prob}^{\mathbf{C}}_{\bot}(s,\phi)$, $\overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X})$ and $\overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X})$ are replaced by approximate values $Prob^{\mathbf{C}}_{\top}(s,\phi)$, $Prob^{\mathbf{C}}_{\bot}(s,\phi)$, $Exp^{\mathbf{C}}_{\top}(s,\mathsf{X})$ and $Exp^{\mathbf{C}}_{\bot}(s,\mathsf{X})$, respectively, that satisfy the following:

$$
\begin{aligned}
Prob^{\mathbf{C}}_{\top}(s,\phi) \geq \overline{Prob}^{\mathbf{C}}_{\top}(s,\phi) \wedge\ Prob^{\mathbf{C}}_{\bot}(s,\phi) \leq \overline{Prob}^{\mathbf{C}}_{\bot}(s,\phi) \\
Exp^{\mathbf{C}}_{\top}(s,\mathsf{X}) \geq \overline{Exp}^{\mathbf{C}}_{\top}(s,\mathsf{X}) \wedge\ Exp^{\mathbf{C}}_{\bot}(s,\mathsf{X}) \leq \overline{Exp}^{\mathbf{C}}_{\bot}(s,\mathsf{X}) \text{ for } \mathsf{X} \in \{\mathsf{X}_{\mathsf{I}^{=t}}, \mathsf{X}_{\mathsf{C}^{\leq t}}\}.
\end{aligned}
$$

Since we get that $\forall s \in \mathbb{S} : s \in Sat^{\bot}_{\mathbf{C}}(\Phi) \Rightarrow s \models^{*}_{\bot} \Phi \Rightarrow s \models_{\bot} \Phi \Rightarrow s \in \overline{Sat}^{\bot}_{\mathbf{C}}(\Phi)$ and also $s \in \overline{Sat}^{\top}_{\mathbf{C}}(\Phi) \Rightarrow s \models_{\top} \Phi \Rightarrow s \models^{*}_{\top} \Phi \Rightarrow s \in Sat^{\top}_{\mathbf{C}}(\Phi)$, the sets $Sat^{\top}_{\mathbf{C}}(\Phi)$ and $Sat^{\bot}_{\mathbf{C}}(\Phi)$ give us the correct approximations of the sets $\overline{Sat}^{\top}_{\mathbf{C}}(\Phi)$ and $\overline{Sat}^{\bot}_{\mathbf{C}}(\Phi)$, i.e., $\overline{Sat}^{\top}_{\mathbf{C}}(\Phi) \subseteq Sat^{\top}_{\mathbf{C}}(\Phi)$ and $Sat^{\bot}_{\mathbf{C}}(\Phi) \subseteq \overline{Sat}^{\bot}_{\mathbf{C}}(\Phi)$.

In contrast to the exact values their approximations can be efficiently computed using the parametrized uniformization. Therefore, we can also effectively obtain the approximated sets $Sat^{\top}_{\mathbf{C}}(\Phi)$ and $Sat^{\bot}_{\mathbf{C}}(\Phi)$ that are further used in the computation of the

min-max approximation. Formally $min_s^{\Phi,\mathbf{P}} = Prob_{\bot}^{\mathbf{C}}(s,\phi)$ and $max_s^{\Phi,\mathbf{P}} = Prob_{\top}^{\mathbf{C}}(s,\phi)$ if the topmost operator of the formula Φ is $\mathsf{P}_{=?}[\phi]$. Similarly, $min_s^{\Phi,\mathbf{P}} = Exp_{\bot}^{\mathbf{C}}(s,\mathsf{X})$ and $max_s^{\Phi,\mathbf{P}} = Exp_{\top}^{\mathbf{C}}(s,\mathsf{X})$ for $\mathsf{X} = \mathsf{X}_{\mathsf{C}\leq t}$ and $\mathsf{X} = \mathsf{X}_{\mathsf{I}=t}$ if the topmost operator of the formula Φ is $\mathsf{R}_{=?}[\mathsf{C}^{\leq t}]$ and $\mathsf{R}_{=?}[\mathsf{I}^{=t}]$, respectively.

5 Parametrized Uniformization

The most important step of the proposed min-max approximation is for each state s to compute the values $Prob_{\top}^{\mathbf{C}}(s,\phi)$, $Prob_{\bot}^{\mathbf{C}}(s,\phi)$, $Exp_{\top}^{\mathbf{C}}(s,\mathsf{X})$ and $Exp_{\bot}^{\mathbf{C}}(s,\mathsf{X})$ where $\mathbf{C}$ is an infinite set of parametrized CTMCs, ϕ is an arbitrary path formula and $\mathsf{X} \in \{\mathsf{X}_{\mathsf{C}\leq t}, \mathsf{X}_{\mathsf{I}=t}\}$. In order to efficiently obtain these values we employ parametrized uniformization. It is a technique that for the given set $\mathbf{C}$, state $s \in \mathbb{S}$ and time $t \in \mathbb{R}_{\geq 0}$ computes vectors $\pi_{\top}^{\mathbf{C},s,t}$ and $\pi_{\bot}^{\mathbf{C},s,t}$ such that for each state $s' \in \mathbb{S}$ the following holds:

$$\pi_{\top}^{\mathbf{C},s,t}(s') \geq max\{\pi^{\mathcal{C}_p,s,t}(s') \mid \mathcal{C}_p \in \mathbf{C}\} \ \wedge \ \pi_{\bot}^{\mathbf{C},s,t}(s') \leq min\{\pi^{\mathcal{C}_p,s,t}(s') \mid \mathcal{C}_p \in \mathbf{C}\} \quad (1)$$

The key idea of parametrized uniformization is to modify standard uniformization in such a way that for each state s' and in each iteration i of the computation we locally minimize (maximize) the value $\pi^{\mathbf{C},s,t}(s')$ with respect to each $\mathcal{C}_p \in \mathbf{C}$. It means that in the ith iteration of the computation for a state s' we consider only the minimal (maximal) values of the relevant states in the iteration $i-1$, i.e., the states that affect the value of state s'. We show that the local minimum and maximum can be efficiently computed and that it gives us values satisfying Equation 1.

Let $\mathbf{Q}^{\mathsf{unif}(\mathbf{C})}$ be an uniformized infinitesimal generator matrix for a set $\mathbf{C}$ of parametrized CTMCs defined as follows:

$$\mathbf{Q}^{\mathsf{unif}(\mathbf{C})}(i,j) = \begin{cases} \sum_{r \in \mathsf{reac}(\mathbf{X}_i,\mathbf{X}_j)} k_r \cdot \frac{C_{r,i}}{q_{max}} & \text{if } i \neq j \\ 1 - \sum_{l \neq i} \sum_{r \in \mathsf{reac}(\mathbf{X}_i,\mathbf{X}_l)} k_r \cdot \frac{C_{r,i}}{q_{max}} & \text{otherwise.} \end{cases} \quad (2)$$

where $q_{max} \geq E_{max} = max\{E^{\mathcal{C}_p}(s) \mid \mathcal{C}_p \in \mathbf{C}, s \in \mathbb{S}\}$ and k_r is a variable from $[k_r^{\bot}, k_r^{\top}]$.

For sake of simplicity, we present only the method allowing to efficiently compute the vector $\pi_{\top}^{\mathbf{C},s,t}$, since the computation of $\pi_{\bot}^{\mathbf{C},s,t}$ is symmetric. We start with the trivial observation that vectors $\pi^{\mathcal{C}_p,s,0}$ (initial probability distributions, e.g., $\pi^{\mathcal{C}_p,s,0}(s') = 1$, if $s = s'$, and 0, otherwise) are equal for all $\mathcal{C}_p \in \mathbf{C}$. Therefore $\pi^{\mathbf{C},s,0} = \pi_{\top}^{\mathbf{C},s,0} = \pi^{\mathcal{C}_p,s,0}$ for all $\mathcal{C}_p \in \mathbf{C}$. In order to present parametrized uniformization, we introduce an operator $\odot_{\top}$ such that for each $s' \in \mathbb{S}$ the following holds:

$$\left(\pi_{\top}^{\mathbf{C},s,0} \odot_{\top} \left(\mathbf{Q}^{\mathsf{unif}(\mathbf{C})}\right)^i\right)(s') \geq max\left\{\left(\pi^{\mathcal{C}_p,s,0} \cdot \left(\mathbf{Q}^{\mathsf{unif}(\mathcal{C}_p)}\right)^i\right)(s') \mid \mathcal{C}_p \in \mathbf{C}\right\}.$$

Moreover, we further require that vectors from the previous iteration can be used, in particular, $\pi_{\top}^{\mathbf{C},s,0} \odot_{\top} (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^i = \left(\pi_{\top}^{\mathbf{C},s,0} \odot_{\top} (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^{i-1}\right) \odot_{\top} \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}$. The operator $\odot_{\top}$ returns a vector $\pi'_{\top} \in \mathbb{R}_{\geq 0}^{|\mathbb{S}|}$ containing for each state $s_i \in \mathbb{S}$ the maximal possible probability after a single discrete step of a DTMC obtained by uniformization of any CTMC

$\mathcal{C}_p$, i.e., $\left(\pi \odot_\top \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}\right)(s) \stackrel{def}{=} max\left\{\left(\pi \cdot \mathbf{Q}^{\mathsf{unif}(\mathcal{C}_p)}\right)(s) \mid p \in \mathbf{P}\right\} = \pi'_\top(s)$ where π is a general vector. Since $\sum \pi'_\top(i) \geq 1$, the vector $\pi'_\top$ is no longer a state distribution.

To show how the operator $\odot_\top$ is computed let $\sigma(s_i)$ be an algebraic expression defined as the part of the vector-matrix multiplication for state s_i. For each $s_i \in \mathbb{S}$ we get $\sigma(s_i) = \left(\pi \cdot \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}\right)(s_i) = \sum_{j=0}^{|\mathbb{S}|-1} \pi(j) \cdot \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}(j,i)$. Rewriting $\sigma(s_i)$ by Equation 2 and using the sets $\mathsf{pred}(s_i)$ and $\mathsf{succ}(s_i)$ we obtain the following:

$$\sigma(s_i) = \sum_{(j,r)\in \mathsf{pred}(s_i)} \pi(j) \cdot k_r \cdot \frac{C_{r,j}}{q_{max}} + \pi(i)\left(1 - \sum_{(j,r)\in \mathsf{succ}(s_i)} k_r \cdot \frac{C_{r,i}}{q_{max}}\right) \tag{3}$$

The first summand in Equation 3, indexed over predecessors of s_i, corresponds to the probability mass inflowing into state s_i through all reactions. The second summand corresponds to the portion of probability mass remaining in s_i from the previous iteration.

The operator $\odot_\top$ locally maximizes expression $\sigma(s)$ for all $s \in \mathbb{S}$ with respect to $\mathbf{P}$, i.e., $\left(\pi \odot_\top \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}\right)(s) = max\{\sigma_p(s) \mid p \in \mathbf{P}\}$ where $\sigma_p(s)$ is the evaluation of $\sigma(s)$ in the parameter point $p = (k_{1_p}, \ldots k_{M_p})$. First, we show that to compute expression $\pi_\top^{\mathbf{C},s,0} \odot_\top (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^i$ it is sufficient to consider only maximal values from the previous iteration, i.e., vector $\pi_\top^{\mathbf{C},s,0} \odot_\top (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^{i-1}$. Note that $\forall s_i \in \mathbb{S}.\ \pi(i) \geq 0$ and $\forall (j,r) \in \mathsf{pred}(s_i) \cup \mathsf{succ}(s_i).\ k_r \geq 0 \ \wedge\ C_{r,j} \geq 0$. Moreover, since $q_{max} \geq E_{max} \geq 0$, we get that $(1 - \sum_{(j,r)\in \mathsf{succ}(s_i)} k_r \cdot \frac{C_{r,i}}{q_{max}}) \geq (1 - \frac{E_{max}}{q_{max}}) \geq 0$. Now, we can see from Equation 3 that in order to maximize $\sigma(s_i)$ maximal values of $\pi(i)$ for each $0 \leq i < |\mathbb{S}|$ have to be taken.

Second, we show how to determine $p = \{k_1, \ldots, k_M\} \in \mathbf{P}$ such that $\sigma(s_i)$ is evaluated as a maximum. Equation 3 can be rewritten in the following way:

$$\sigma(s_i) = \sum_{(j,r)\in \mathsf{in}} k_r \cdot \frac{\pi(j) \cdot C_{r,j}}{q_{max}} + \sum_{(j,r)\in \mathsf{inout}} k_r \cdot \frac{\pi(j) \cdot C_{r,j} - \pi(i) \cdot C_{r,i}}{q_{max}} - \sum_{(j,r)\in \mathsf{out}} k_r \cdot \frac{\pi(i) \cdot C_{r,i}}{q_{max}}$$

where $\mathsf{in} = \mathsf{pred}(s_i) \setminus \mathsf{succ}(s_i), \mathsf{inout} = \mathsf{pred}(s_i) \cap \mathsf{succ}(s_i)$ and $\mathsf{out} = \mathsf{succ}(s_i) \setminus \mathsf{pred}(s_i)$. The three sums range over disjoint sets of reactions. The first sum represents all incoming reactions that do not have an outgoing counterpart, for these $k_r = k_r^\top$, since they only increase $\sigma(s_i)$. The second sum represents reactions flowing into s_i as well as flowing out of s_i. In this case, expression $\pi(j) \cdot C_{r,j} - \pi(i) \cdot C_{r,i}$ has to be evaluated. If it is positive, $k_r = k_r^\top$, otherwise, $k_r = k_r^\bot$. The last sum represents only reactions flowing out of s_i and hence $k_r = k_r^\bot$. The operator $\odot_\top$ is now computed as $\left(\pi \odot_\top \mathbf{Q}^{\mathsf{unif}(\mathbf{C})}\right)(s_i) = \sigma(s_i)$ where each k_r inside $\sigma(s_i)$ is chosen according to the aforementioned rules.

The computation of $\pi_\bot^{\mathbf{C},s,t}$ is symmetric to the case of $\pi_\top^{\mathbf{C},s,t}$. It means that we define the operator $\odot_\bot$ which locally minimize expression $\sigma(s)$ for all $s \in \mathbb{S}$ with respect to $\mathbf{P}$. In order to minimize $\sigma(s_i)$ it is sufficient to inverse the aforementioned rules.

Vectors $\pi_\top^{\mathbf{C},s,t}$ and $\pi_\bot^{\mathbf{C},s,t}$ are now computed similarly as in the case of standard uniformization, i.e., $\pi_\star^{\mathbf{C},s,t} = \sum_{i=L_\epsilon}^{R_\epsilon} \gamma_{i,q\cdot t} \cdot \pi^{\mathbf{C},s,0} \odot_\star (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^i$ where $\star \in \{\bot, \top\}$. To obtain the required values $Prob_\top^{\mathbf{C}}(s,\phi)$, $Prob_\bot^{\mathbf{C}}(s,\phi)$, $Exp_\top^{\mathbf{C}}(s,\mathsf{X})$ and $Exp_\bot^{\mathbf{C}}(s,\mathsf{X})$, we employ the standard model checking technique [3,21] where transient probability $\pi^{\mathcal{C},s,t}$ for a non-parametrized CTMC $\mathcal{C}$ is replaced by vectors $\pi_\top^{\mathbf{C},s,t}$ and $\pi_\bot^{\mathbf{C},s,t}$.

Compared to standard uniformization, only a constant amount of additional work has to be performed in order to determine parameter values. Therefore, asymptotic complexity of parametrized uniformization remains the same as standard uniformization.

6 Parameter Space Decomposition

Before we describe parameter space decomposition – a method allowing to reduce the inaccuracy of the proposed min-max approximation – we briefly discuss the key characteristics of parametrized uniformization. The most important fact is that parametrized uniformization for the set $\mathbf{C}$ in general does not correspond to standard uniformization for any CTMC $\mathcal{C}_p \in \mathbf{C}$. The reason is that we consider a behaviour of a parametrized CTMC that has no equivalent counterpart in any particular $\mathcal{C}_p$. First, the parameter k_r in Equation 3 is determined individually for each state. Therefore, in a single iteration $k_r = k_r^{\top}$ for one state and $k_r = k_r^{\bot}$ for another state. Second, the parameter is determined individually for each iteration and thus for a state s_i the parameter k_r can be chosen differently in individual iterations.

Inaccuracy of the proposed min-max approximation related to the computation of parametrized uniformization, called *unification error*, is given as $(max_s^{\Phi,\mathbf{P}} - \overline{max}_s^{\Phi,\mathbf{P}}) + (\overline{min}_s^{\Phi,\mathbf{P}} - min_s^{\Phi,\mathbf{P}})$. Apart from the unification error our approach introduces an inaccuracy related to approximation of the landscape function $\lambda_s^{\Phi,\mathbf{P}_i}$, called *approximation error*, given as $\overline{max}_s^{\Phi,\mathbf{P}} - \overline{min}_s^{\Phi,\mathbf{P}}$. Finally, the *overall error* of the min-max approximation, denoted as $\mathsf{Err}_s^{\Phi,\mathbf{P}}$, is defined as a sum of both errors, i.e., $\mathsf{Err}_s^{\Phi,\mathbf{P}} = max_s^{\Phi,\mathbf{P}} - min_s^{\Phi,\mathbf{P}}$. Fig. 1 illustrates both types of errors. The approximation error is depicted as blue rectangles and the unification error is depicted as the red rectangles.

We are not able to effectively distinguish the proportion of the approximation error and the unification error nor to reduce the unification error as such. Therefore, we design a method based on the parameter space decomposition that allows us to effectively reduce the overall error of the min-max approximation to a user specified *absolute error bound*, denoted as ERR.

In order to ensure that the min-max approximation meets the given absolute error bound ERR, we iteratively decompose the parameter space $\mathbf{P}$ into finitely many subspaces such that $\mathbf{P} = \mathbf{P}_1 \cup \ldots \cup \mathbf{P}_n$ and each partial result satisfies the overall error bound, i.e., $\forall s \in \mathbb{S} : max_s^{\Phi,\mathbf{P}_i} - min_s^{\Phi,\mathbf{P}_i} \leq \text{ERR}$. Therefore, the overall error for each state $s \in \mathbb{S}$ equals to $\mathsf{Err}_s^{\Phi,\mathbf{P}} = \sum_{i=1}^{n} \frac{|\mathbf{P}_i|}{|\mathbf{P}|} \left(max_s^{\Phi,\mathbf{P}_i} - min_s^{\Phi,\mathbf{P}_i} \right) \leq \sum_{i=1}^{n} \frac{|\mathbf{P}_i|}{|\mathbf{P}|} \text{ERR} = \text{ERR}$. Fig. 1 illustrates such a decomposition and demonstrates convergence of $\mathsf{Err}_s^{\Phi,\mathbf{P}_i}$ to 0 provided that the function $\lambda_s^{\Phi,\mathbf{P}_i}$ is continuous.

For sake of simplicity, we present parametric decomposition on the computation of $\pi_{\top}^{\mathbf{C},s,t}$ since it can be easily extended to the computation of $Prob_{\top}^{\mathbf{C}}(s,\phi)$, $Prob_{\bot}^{\mathbf{C}}(s,\phi)$, $Exp_{\top}^{\mathcal{C}}(s,\mathsf{X})$ and $Exp_{\bot}^{\mathcal{C}}(s,\mathsf{X})$. If during the computation in an iteration i for a state $s' \in \mathbb{S}$ holds that $\left(\pi_{\top}^{\mathbf{C},s,0} \odot_{\top} (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^i\right)(s') - \left(\pi_{\bot}^{\mathbf{C},s,0} \odot_{\bot} (\mathbf{Q}^{\mathsf{unif}(\mathbf{C})})^i\right)(s') > \text{ERR}$ we cancel the current computation and decompose the parameter space $\mathbf{P}$ to n subspaces such that $\mathbf{P} = \mathbf{P}_1 \cup \ldots \cup \mathbf{P}_n$. Each subspace $\mathbf{P}_j$ defines a new set of CTMCs $\mathbf{C}_j = \{\mathcal{C}_j \mid j \in \mathbf{P}_j\}$ that is independently processed in a new computation branch. Note that we could

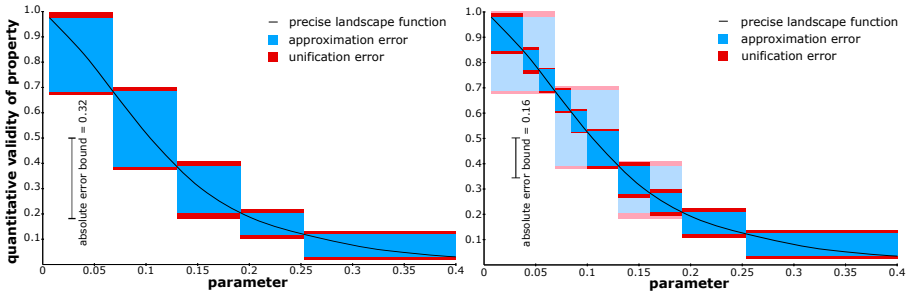


Fig. 1. Illustration of the min-max approximation computation of the landscape function $\lambda_s^{\Phi,\mathbf{P}}$ for an initial state s, property Φ and parameter space $\mathbf{P} = [0, 0.4]$. Left graph shows the decomposition of $\mathbf{P}$ into 5 subspaces for absolute error bound ERR = 0.32. Right graph shows a more refined decomposition for ERR = 0.16 resulting in 10 subspaces. This decomposition reduces both types of errors in each refined subspaces. The exact shape of $\lambda_s^{\Phi,\mathbf{P}}$ is visualized as the black curve.

reuse the previous computation and continue from the iteration $i-1$. However, the most significant part of the error is usually cumulated during the the previous iterations and thus the decomposition would have only a negligible impact on error reduction.

A minimal decomposition with respect to the parameter space $\mathbf{P}$ defines a minimal number of subspaces m such that $\mathbf{P} = \mathbf{P}_1 \cup \ldots \cup \mathbf{P}_m$ and for each subspace $\mathbf{P}_j$ where $1 \leq j \leq m$ holds that $\mathrm{Err}_s^{\Phi,\mathbf{P_j}} \leq \mathrm{ERR}$. Note that the existence of such decomposition is guaranteed only if the landscape function $\lambda_s^{\Phi,\mathbf{P}}$ is continuous. If the landscape function is continuous there can exist more than one minimal decomposition. However, it can not be straightforwardly found. To overcome this problem we have considered and implemented several heuristics allowing to iteratively compute a decomposition satisfying the following: (1) it ensures the required error bound whenever $\lambda_s^{\Phi,\mathbf{P}}$ is continuous, (2) it guarantees the refinement termination in the situation where $\lambda_s^{\Phi,\mathbf{P}}$ is not continuous and the discontinuity causes that ERR can not be achieved. To ensure the termination an additional parameter has to be introduced as a lower bound on the subspace size. Hence this parameter provides a supplementary termination criterion.

7 Case Studies

We implemented our method on top of the tool PRISM 4.0 [20]. We run all experiments on a Linux workstation with an AMD Phenom™ II X4 940 Processor @ 3GHz, 8 GB DDR2 @ 1066 MHz RAM. We used PRISM version 4.0.3. running with sparse engine, since this engine is typically faster than its symbolic counterparts due to efficient matrix vector multiplication.

Schloegl's Model. We use Schlogel's model [24] to demonstrate the practicability of our method for parameter exploration with respect to basic transient analysis. It is the simplest biochemical reaction model for which stochasticity is crucial due to bi-stability

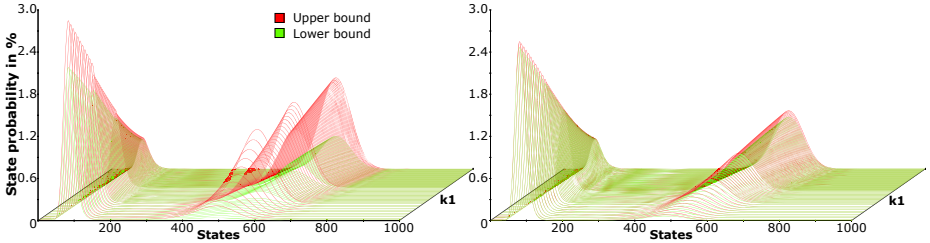


Fig. 2. Species X distribution at 20 time units for $k_1 \in [0.029, 0.031]$ (in s^{-1}). The two presented cases differ in absolute error bound: (left) ERR = 0.01, (right) ERR = 0.001.

– existence of two different steady states to which the species population can (nondeterministically) converge. The model is defined by the following reactions [8]: $2X \xrightarrow{k_1} 3X$, $3X \xrightarrow{k_2} 2X$, $\emptyset \xrightarrow{k_3} X$, $X \xrightarrow{k_4} \emptyset$; $k_1 = 0.03s^{-1}, k_2 = 10^{-4}s^{-1}, k_3 = 200s^{-1}, k_4 = 3.5s^{-1}$. Deterministic formulation of the model by means of ordinary differential equations (ODE) predicts for $k_1 \in [0.0285, 0.035]$ two steady states to which the population converges in the horizon of 20 time units. We will focus on the range $k_1 \in [0.029, 0.031]$. Under the deterministic setting, from any initial state the dynamics evolves to a single steady state. In the noisy setting [26], the population of molecules distributes around both steady states (in short time perspective, here 20 time units). In long time perspective, the population oscillates around both steady states.

We focus on the short time-scale – to analyze the population of X at time 20 starting from the initial state where the number of X is 250. According to the respective ODE model, the population always converges to an asymptotic steady state $X_{st} \leq 1000$. Considering this as an assumption it allows us to bound the state space. The corresponding CTMC has 1001 states and 2000 transitions. The goal of the analysis is to explore how the observed distribution is affected when perturbing k_1 in the range $[0.029, 0.031]$. By executing our method for the absolute error bound ERR $= 0.01$ we got the result visualized in Fig. 2 (left). It can be directly seen that for each parameter point there is a non-zero probability that some individuals reside near the higher steady state while some reside near the lower steady state at time 20. The jumps that are mostly observable in distributions around the higher steady state are caused by the approximation error. Computation with a one order lower error gives a smooth result, see Fig. 2 (right).

The computation required $7.36 \cdot 10^5$ iterations of the parametrized uniformization. The parameter decomposition resulted in 76 subspaces for ERR $= 0.01$ and 639 subspaces for ERR $= 0.001$. The overall computation took 2 and 16.5 hours, respectively.

Gene Regulation of Mammalian Cell Cycle. We have applied the min-max approximation to the gene regulation model published in [25], the regulatory network is shown in Fig. 3a. The model explains regulation of a transition between early phases of the mammalian cell cycle. In particular, it targets the transition from the control G_1-phase to S-phase (the synthesis phase). G_1-phase makes an important checkpoint controlled by a *bistable regulatory circuit* based on an interplay of the retinoblastoma protein *pRB*, denoted by A (the so-called tumour suppressor, HumanCyc:HS06650) and the

(a)

Gene a interactions		Gene b interactions	
$a \to a+A$	1	$b \to b+B$	0.05
$aB \to aB+A$	1	$bB \to bB+B$	1
$A+a \leftrightarrow aA$	100; 10	$A+b \leftrightarrow bA$	100; 10
$B+a \leftrightarrow aB$	100; 10	$B+b \leftrightarrow bB$	100; 10
Protein degradation			
$A \to$	γ_A	$B \to$	γ_B

(b)

Property	# iter.	# subsp.	time[h]
(1a)	$1.2{\cdot}10^6$	153	9
(2a)	$2.0{\cdot}10^6$	69	5.5
(3a)	$2.0{\cdot}10^6$	66	4.5
(1b)	$4.0{\cdot}10^6$	159	10.5
(2b)	$4.0{\cdot}10^6$	132	8
(3b)	$4.0{\cdot}10^6$	80	5

(c)

Fig. 3. (a) Two-gene regulatory circuit controlling G_1/S transition in mammalian cell cycle. (b) Stochastic mass action model of the G_1/S regulatory circuit – a,b represent genes, aA, aB, bA, bB represent transcription factor-gene promoter complexes (c) Computation results.

retinoblastoma-binding transcription factor E_2F_1, denoted by B (a central regulator of a large set of human genes, HumanCyc:HS02261). In high concentration levels, the E_2F_1 protein activates the G_1/S transition mechanism. On the other hand, a low concentration of E_2F_1 prevents committing to S-phase.

Positive autoregulation of B causes bi-stability of its concentration depending on the parameters. Especially, of specific interest is the degradation rate of A, γ_A. In [25] it is shown that for increasing γ_A the low stable mode of B switches to the high stable mode. When mitogenic stimulation increases under conditions of active growth, rapid phosphorylation of A starts and makes the degradation of unphosphorylated A stronger (the degradation rate γ_A increases). This causes B to lock in the high stable mode implying the cell cycle commits to S-phase. Since mitogenic stimulation influences the degradation rate of A, our goal is to study the population distribution around the low and high steady state and to explore the effect of γ_A by means of the landscape function.

We have translated the original ODE model into the framework of stochastic mass action kinetics [12]. The resulting reactions are shown in Fig. 3b. Since the detailed knowledge of elementary chemical reactions occurring in the process of transcription and translation is incomplete, we use the simplified form as suggested in [10]. In the minimalistic setting, the reformulation requires addition of rate parameters describing the transcription factor–gene promoter interaction while neglecting cooperativeness of transcription factors activity. Our parametrization is based on time-scale orders known for the individual processes [27] (parameters considered in s^{-1}). Moreover, we assume the numbers of A and B are bounded by 10 molecules. Upper bounds for A and B are set with respect to behaviour of an ensemble of stochastic simulations. We consider minimal population number distinguishing the two stable modes. All other species are bounded by the initial number of DNA molecules (genes a and b) which is conserved and set to 1. The corresponding CTMC has 1078 states and 5919 transitions.

We consider three hypotheses: (1) stabilization in the low mode where $B < 3$, (2) stabilization in the high mode where $B > 5$, (3) stabilization in the high mode where $B > 7$ ((3) is more focused than (2)). All the hypotheses are expressed within time horizon 1000 seconds reflecting the time scale of gene regulation response. We employ two alternative CSL formulations to express each of the three hypothesis. According to [25], we consider the parameter space $\gamma_A \in [0.005, 0.5]$.

First, we express the property of being inside the given bound during the time interval $I = [500, 1000]$ using globally operator: (1a) $\mathsf{P}_{\sim ?}[\mathsf{G}^I(B < 3)]$, (2a) $\mathsf{P}_{\sim ?}[\mathsf{G}^I(B > 5)]$ and $(3a)$ $\mathsf{P}_{\sim ?}[\mathsf{G}^I(B > 7)]$. The interval starts from 500 seconds in order to bridge the

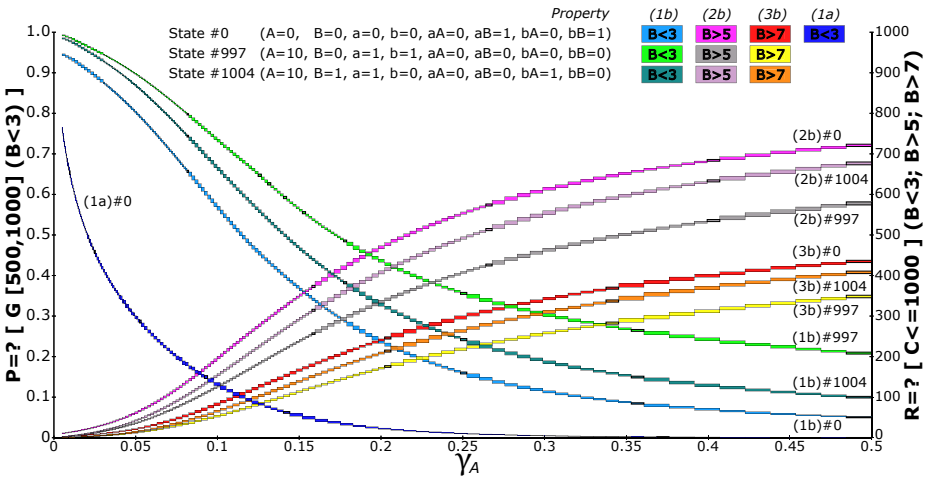


Fig. 4. Landscape functions of properties *(1a,1b,2b,3b)* for $\gamma_A \in [0.005, 0.5]$ (in s^{-1}) and initial states #0, #997 and #1004. The left Y-axis scale corresponds to *(1a)*, the right to *(1b,2b,3b)*.

initial fluctuation region and let the system stabilize. Since the stochastic noise causes molecules to repeatedly escape the requested bound, the resulting probability is significantly lower than expected. Namely, in cases (2a) and (3a) the resulting probability is close to 0 for the whole parameter space. Moreover, the selection of an initial state has only a negligible impact on the result. Therefore, in Fig. 4 only the resulting probability for case (1a) and a single selected initial state is visualized.

Second, we use a cumulative reward property to capture the fraction of the time the system has the required number of molecules within the time interval $[0, 1000]$: $(1b)$ $\mathsf{R}_{\sim ?}[\mathsf{C}^{\leq t}](B < 3)$, $(2b)$ $\mathsf{R}_{\sim ?}[\mathsf{C}^{\leq t}](B > 5)$, $(3b)$ $\mathsf{R}_{\sim ?}[\mathsf{C}^{\leq t}](B > 7)$ where $t = 1000$ and $\mathsf{R}_{\sim ?}[\mathsf{C}^{\leq t}](B \sim X)$ denotes that state reward ρ is defined such that $\forall s \in \mathbb{S}.\rho(s) = 1$ iff $B \sim X$ in s. The result is visualized for three selected initial states in Fig. 4.

Fig. 4 also illustrates inaccuracy of our approach with respect to the absolute error bound $\textsc{Err} = 0.01$ by means of small rectangles depicting approximations of the resulting probabilities and expected rewards. The analyses predict that the distribution of the low steady mode interferes with the distribution of the high steady mode. It confirms bi-stability predicted in [25] but in contrast to ODE analysis our method shows how the population of cells distributes around the two stable states. Results of computations including the number of iterations performed during parametrized uniformization, numbers of resulting subspaces and execution times in hours, are presented in Fig. 3c.

Finally, to see how degradation rates of A and B cooperate in affecting property $(3b)$, we explore two-dimensional parameter space $(\gamma_A, \gamma_B) \in [0.005, 0.1] \times [0.05, 0.1]$. The computation also required $4.0 \cdot 10^6$ iterations of the parametrized uniformization, the parameter decomposition resulted in 143 subspaces for $\textsc{Err} = 0.1$ and the overall execution took 14 hours. Fig. 5 illustrates the computed upper bound of the landscape function for initial state #0 and the absolute error. The result predicts antagonistic relation between the degradation rates which is in agreement with the ODE model [25].

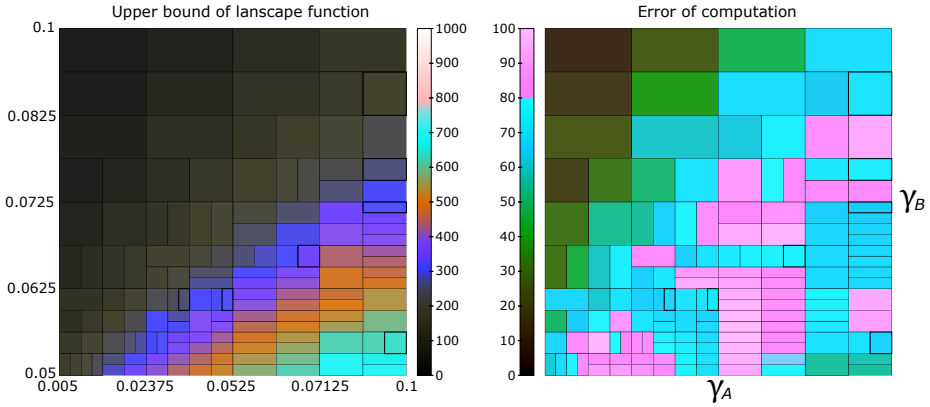


Fig. 5. Landscape function for property $(3b)$, initial state #0 $(A = 0, B = 0, a = 0, b = 0, aA = 0, aB = 1, bA = 0, bB = 1)$ and two-dimensional parameter space $(\gamma_A, \gamma_B) \in [0.005, 0.1] \times [0.05, 0.1]$ (represented in s^{-1} by X and Y axes, respectively). On the left, the upper bound of the landscape function is illustrated. On the right, the absolute error given as difference between computed upper and lower bounds is depicted. In both cases the color scale is used.

8 Conclusions

We have introduced the parameter exploration problem for stochastic biochemical systems as the computation of a landscape function for a given temporal logic formula. The key idea of our approach is to approximate the lower and upper bounds of the landscape function. To obtain such approximation for an arbitrary nested CSL formula, we compute the largest and smallest set of states satisfying the formula using parametrized uniformization. This allows to approximate the minimal and maximal transient probability with respect to the parameter space. In order to reach a required error bound of the proposed approximation, we iteratively decompose the parameter space and compute the approximation for each subspace individually. We have demonstrated our approach to the parameter exploration problem on two biologically motivated case studies.

The experiments show that our method can be extremely time demanding and thus in our future work we will focus on its acceleration. We plan to apply techniques allowing to accelerate the underlying transient analysis [9,16] and more efficient heuristics for the parameter space decomposition. Moreover, our method can be easily parallelized and thus a significant acceleration can be obtained.

References

1. Andreychenko, A., Mikeev, L., Spieler, D., Wolf, V.: Parameter Identification for Markov Models of Biochemical Reactions. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 83–98. Springer, Heidelberg (2011)
2. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.: Verifying Continuous Time Markov Chains. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 269–276. Springer, Heidelberg (1996)

3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.P.: Model Checking Continuous-Time Markov Chains by Transient Analysis. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 358–372. Springer, Heidelberg (2000)
4. Ballarini, P., Forlin, M., Mazza, T., Prandi, D.: Efficient Parallel Statistical Model Checking of Biochemical Networks. In: PDMC 2009. EPTCS, vol. 14, pp. 47–61 (2009)
5. Barbuti, R., Levi, F., Milazzo, P., Scatena, G.: Probabilistic Model Checking of Biological Systems with Uncertain Kinetic Rates. Theor. Comput. Sci. 419, 2–16 (2012)
6. Bernardini, F., Biggs, C., Derrick, J., Gheorghe, M., Niranjan, M., Sanguinetti, G.: Parameter Estimation and Model Checking in a Model of Prokaryotic Autoregulation. Tech. rep., University of Sheffield (2007)
7. Daigle, B., Roh, M., Petzold, L., Niemi, J.: Accelerated Maximum Likelihood Parameter Estimation for Stochastic Biochemical Systems. BMC Bioinformatics 13(1), 68–71 (2012)
8. Degasperi, A., Gilmore, S.: Sensitivity Analysis of Stochastic Models of Bistable Biochemical Reactions. In: Bernardo, M., Degano, P., Zavattaro, G. (eds.) SFM 2008. LNCS, vol. 5016, pp. 1–20. Springer, Heidelberg (2008)
9. Didier, F., Henzinger, T.A., Mateescu, M., Wolf, V.: Fast Adaptive Uniformization of the Chemical Master Equation. In: HIBI 2009, pp. 118–127. IEEE Computer Society (2009)
10. El Samad, H., Khammash, M., Petzold, L., Gillespie, D.: Stochastic Modelling of Gene Regulatory Networks. Int. J. of Robust and Nonlinear Control 15(15), 691–711 (2005)
11. Fox, B.L., Glynn, P.W.: Computing Poisson Probabilities. CACM 31(4), 440–445 (1988)
12. Gillespie, D.T.: Exact Stochastic Simulation of Coupled Chemical Reactions. Journal of Physical Chemistry 81(25), 2340–2381 (1977)
13. Golightly, A., Wilkinson, D.J.: Bayesian Parameter Inference for Stochastic Biochemical Network Models Using Particle Markov Chain Monte Carlo. Interface Focus 1(6), 807–820 (2011)
14. Grassmann, W.: Transient Solutions in Markovian Queueing Systems. Computers & Operations Research 4(1), 47–53 (1977)
15. Hahn, E.M., Han, T., Zhang, L.: Synthesis for PCTL in Parametric Markov Decision Processes. In: NASA Formal Methods, pp. 146–161 (2011)
16. Henzinger, T.A., Mateescu, M., Wolf, V.: Sliding Window Abstraction for Infinite Markov Chains. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 337–352. Springer, Heidelberg (2009)
17. Jha, S.K., Clarke, E.M., Langmead, C.J., Legay, A., Platzer, A., Zuliani, P.: A Bayesian Approach to Model Checking Biological Systems. In: Degano, P., Gorrieri, R. (eds.) CMSB 2009. LNCS, vol. 5688, pp. 218–234. Springer, Heidelberg (2009)
18. Koh, C.H., Palaniappan, S., Thiagarajan, P., Wong, L.: Improved Statistical Model Checking Methods for Pathway Analysis. BMC Bioinformatics 13(suppl. 17), S15 (2012)
19. Kwiatkowska, M., Norman, G., Pacheco, A.: Model Checking Expected Time and Expected Reward Formulae with Random Time Bounds. Compu. Math. Appl. 51(2), 305–316 (2006)
20. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
21. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic Model Checking. In: Bernardo, M., Hillston, J. (eds.) SFM 2007. LNCS, vol. 4486, pp. 220–270. Springer, Heidelberg (2007)
22. Mikeev, L., Neuhäußer, M., Spieler, D., Wolf, V.: On-the-fly Verification and Optimization of DTA-properties for Large Markov Chains. Form. Method. Syst. Des., 1–25 (2012)
23. Reinker, S., Altman, R., Timmer, J.: Parameter Estimation in Stochastic Biochemical Reactions. IEEE Proc. Syst. Biol. 153(4), 168–178 (2006)

24. Schlögl, F.: Chemical Reaction Models for Non-Equilibrium Phase Transitions. Zeitschrift fur Physik 253, 147–161 (1972)
25. Swat, M., Kel, A., Herzel, H.: Bifurcation Analysis of the Regulatory Modules of the Mammalian G1/S transition. Bioinformatics 20(10), 1506–1511 (2004)
26. Vellela, M., Qian, H.: Stochastic Dynamics and Non-Equilibrium Thermodynamics of a Bistable Chemical System: the Schlögl Model Revisited. Journal of The Royal Society Interface 6(39), 925–940 (2009)
27. Yang, E., van Nimwegen, E., Zavolan, M., Rajewsky, N., Schroeder, M.K., Magnasco, M., Darnell, J.E.: Decay Rates of Human mRNAs: Correlation With Functional Characteristics and Sequence Attributes. Genome Research 13(8), 1863–1872 (2003)

Parameterized Verification of Asynchronous Shared-Memory Systems

Javier Esparza[1], Pierre Ganty[2,⋆], and Rupak Majumdar[3]

[1] TU Munich
[2] IMDEA Software Institute
[3] MPI-SWS

Abstract. We characterize the complexity of the safety verification problem for parameterized systems consisting of a leader process and arbitrarily many anonymous and identical contributors. Processes communicate through a shared, bounded-value register. While each operation on the register is atomic, there is no synchronization primitive to execute a sequence of operations atomically.

We analyze the complexity of the safety verification problem when processes are modeled by finite-state machines, pushdown machines, and Turing machines. The problem is coNP-complete when all processes are finite-state machines, and is PSPACE-complete when they are pushdown machines. The complexity remains coNP-complete when each Turing machine is allowed boundedly many interactions with the register. Our proofs use combinatorial characterizations of computations in the model, and in case of pushdown-systems, some language-theoretic constructions of independent interest.

1 Introduction

We conduct a systematic study of the complexity of safety verification for *parameterized asynchronous shared-memory systems*. These systems consist of a *leader* process and arbitrarily many identical *contributors*, processes with no identity, running at arbitrarily relative speeds and subject to faults (a process can crash). The shared-memory consists of a read/write register that all processes can access to perform either a read operation or a write operation. The register is bounded: the set of values that can be stored is finite. We do insist that read/write operations execute atomically but sequences of operations do not: no process can conduct an atomic sequence of reads and writes while excluding all other processes. The parameterized verification problem for these systems asks to check if a safety property holds no matter how many contributors are present. Our model subsumes the case in which all processes are identical by having the leader process behave like yet another contributor. The presence of a distinguished leader adds (strict) generality to the problem.

We analyze the complexity of the safety verification problem when leader and contributors are modeled by finite state machines, pushdown machines, and even Turing machines. Using combinatorial properties of the model that allow simulating arbitrarily many contributors using finitely many ones, we show that if leader and contributors are finite-state machines the problem is coNP-complete. The case in which leader and contributors are pushdown machines was first considered by Hague [17], who gave a coNP

⋆ Supported by the Spanish projects with references TIN2010-20639 and TIN2012-39391-C04.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 124–140, 2013.

lower bound and a 2EXPTIME upper bound. We close the gap and prove that the problem is PSPACE-complete. Our upper bound requires several novel language-theoretic constructions on bounded-index approximations of context-free languages. Finally, we address the bounded safety problem, i.e., deciding if no error can be reached by computations in which no contributor nor the leader execute more than a given number k of steps (this does not bound the length of the computation, since the number of contributors is unbounded). We show that (if k is given in unary) the problem is coNP-complete not only for pushdown machines, but also for arbitrary Turing machines. Thus, the safety verification problem when the leader and contributors are poly-time Turing machines is also coNP-complete.

These results show that non-atomicity substantially reduces the complexity of verification. In the atomic case, contributors can ensure that they are the only ones that receive a message: the first contributor that reads the message from the store can also erase it within the same atomic action. This allows the leader to distribute identities to contributors. As a consequence, the safety problem is at least PSPACE-hard for state machines, and undecidable for pushdown machines (in the atomic case, the safety problem of two pushdown machines is already undecidable). A similar argument shows that the bounded safety problem is PSPACE-hard. In contrast, we get several coNP upper bounds, which opens the way to the application of SAT-solving or SMT-techniques.

Besides intellectual curiosity, our work on this model is motivated by practical distributed protocols implemented on wireless sensor networks. In these systems, a central co-ordinator (the base station) communicates with an arbitrary number of mass-produced tiny agents (or motes) that run concurrently and asynchronously. The motes have limited computational power, and for some systems such as vehicular networks anonymity is a requirement [20]. Further, they are susceptible to crash faults. Implementing atomic communication primitives in this setting is expensive and can be problematic: for instance, a process might crash while holding a lock. Thus, protocols in these systems work asynchronously and without synchronization primitives. Our algorithms provide the foundations for safety verification of these systems.

Full proofs of our results can be found in the associated technical report [13].

Related Works. Parameterized verification problems have been extensively studied both theoretically and practically. It is a computationally hard problem: the reachability problem is undecidable even if each process has a finite state space [2]. For this reason, special cases have been extensively studied. They vary according to the main characteristics of the systems to verify like the communication topology of the processes (array, tree, unordered, etc); their communication primitives (shared memory, unreliable broadcasts, (lossy) queues, etc); or whether processes can distinguish from each other (using ids, a distinguished process, etc). Prominent examples include broadcast protocols [12,14,9,8], where finite-state processes communicate via broadcast messages, asynchronous programs [15,23], where finite-state processes communicate via unordered channels, finite-state processes communicating via ordered channels [1], micro architectures [21], cache coherence protocols [10,6], communication protocols [11], multithreaded shared-memory programs [5,7,19,22].

Besides the model of Hague [17], the closest model to ours that has been previously studied [16] is that of distributed computing with identity-free, asynchronous

processors and non-atomic registers. The emphasis there was the development of distributed algorithm primitives such as time-stamping, snapshots, and consensus, using either unbounded registers or an unbounded number of bounded registers.

It was left open if these primitives can be implemented using a bounded number of bounded registers. Our decidability results indicate that this is not possible: the safety verification problem would be undecidable if such primitives could be implemented.

2 Formal Model: Non-atomic Networks

We describe our formal model, called *non-atomic networks*. We take a language-theoretic view, identifying a system with the language of its executions.

Preliminaries. A *labeled transition system* (LTS) is a quadruple $\mathcal{T} = (\Sigma, Q, \delta, q_0)$, where Σ is a finite set of *action labels*, Q is a (non necessarily finite) set of *states*, $q_0 \in Q$ is the *initial state*, and $\delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$ is the *transition relation*, where ε is the empty string. We write $q \xrightarrow{a} q'$ for $(q, a, q') \in \delta$. For $\sigma \in \Sigma^*$, we write $q \xrightarrow{\sigma} q'$ if there exist $q_1, \ldots, q_n \in Q$ and $a_0, \ldots, a_n \in \Sigma \cup \{\varepsilon\}$, $q \xrightarrow{a_0} q_1 \xrightarrow{a_1} q_2 \cdots q_n \xrightarrow{a_n} q'$ such that $a_0 \cdots a_n = \sigma$. The sequence $q \cdots q'$ is called a *path* and σ its *label*. A *trace* of $\mathcal{T}$ is a sequence $\sigma \in \Sigma^*$ such that $q_0 \xrightarrow{\sigma} q$ for some $q \in Q$. Define $L(\mathcal{T})$, the *language* of $\mathcal{T}$, as the set of traces of $\mathcal{T}$. Note that $L(\mathcal{T})$ is *prefix closed*: $L(\mathcal{T}) = \mathit{Pref}(L(\mathcal{T}))$ where $\mathit{Pref}(L) = \{s \mid \exists u\colon s\,u \in L\}$

To model concurrent executions of LTSs, we introduce two operations on languages: the shuffle and the asynchronous product. The *shuffle* of two words $x, y \in \Sigma^*$ is the language $x \sqcup\!\sqcup y = \{x_1 y_1 \ldots x_n y_n \in \Sigma^* \mid \text{each } x_i, y_i \in \Sigma^* \text{and } x = x_1 \cdots x_n \wedge y = y_1 \cdots y_n\}$. The shuffle of two languages L_1, L_2 is the language $L_1 \sqcup\!\sqcup L_2 = \bigcup_{x \in L_1, y \in L_2} x \sqcup\!\sqcup y$. Shuffle is associative, and so we can write $L_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup L_n$ or $\sqcup\!\sqcup_{i=1}^n L_i$.

The *asynchronous product* of two languages $L_1 \subseteq \Sigma_1^*$ and $L_2 \subseteq \Sigma_2^*$, denoted $L_1 \parallel L_2$, is the language L over the alphabet $\Sigma = \Sigma_1 \cup \Sigma_2$ such that $w \in L$ iff the projections of w to Σ_1 and Σ_2 belong to L_1 and L_2, respectively.[1] If a language consists of a single word, e.g. $L_1 = \{w_1\}$, we abuse notation and write $w_1 \parallel L_2$. Asynchronous product is also associative, and so we write $L_1 \parallel \cdots \parallel L_n$ or $\|_{i=1}^n L_i$.

Let $\mathcal{T}_1, \ldots, \mathcal{T}_n$ be LTSs, where $\mathcal{T}_i = (\Sigma_i, Q_i, \delta_i, q_{0i})$. The *interleaving* $\sqcup\!\sqcup_{i=1}^n \mathcal{T}_i$ is the LTS with actions $\bigcup_{i=1}^n \Sigma_i$, set of states $Q_1 \times \cdots \times Q_n$, initial state $(q_{01}, \ldots, q_{0n})$, and a transition $(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')$ iff $(q_i, a, q_i') \in \delta_i$ for some $1 \le i \le n$ and $q_j = q_j'$ for every $j \ne i$. Interleaving models parallel composition of LTSs that do not communicate at all. The language $L(\mathcal{T}_1 \sqcup\!\sqcup \cdots \sqcup\!\sqcup \mathcal{T}_n)$ of the interleaving is $\sqcup\!\sqcup_{i=1}^n L(\mathcal{T}_i)$.

The *asynchronous parallel composition* $\|_{i=1}^n \mathcal{T}_i$ of $\mathcal{T}_1, \ldots, \mathcal{T}_n$ is the LTS having $\bigcup_{i=1}^n \Sigma_i$ as set of actions, $Q_1 \times \cdots \times Q_n$ as set of states, $(q_{01}, \ldots, q_{0n})$ as initial state, and a transition $(q_1, \ldots, q_n) \xrightarrow{a} (q_1', \ldots, q_n')$ if and only if

1. $a \ne \varepsilon$ and for all $1 \le i \le n$ either $a \notin \Sigma_i$ and $q_i = q_i'$ or $a \in \Sigma_i$ and $(q_i, a, q_i') \in \delta_i$, or;
2. $a = \varepsilon$, and there is $1 \le i \le n$ such that $(q_i, \varepsilon, q_i') \in \delta_i$ and $q_j = q_j'$ for every $j \ne i$.

Asynchronous parallel composition models the parallel composition of LTSs in which an action a must be simultaneously executed by every LTSs having a in its alphabet. $L(\mathcal{T}_1 \parallel \cdots \parallel \mathcal{T}_n)$, the language of the asynchronous parallel composition, is $\|_{i=1}^n L(\mathcal{T}_i)$.

[1] Observe that the $L_1 \parallel L_2$ depends on L_1, L_2 **and** also their underlying alphabet Σ_1 and Σ_2.

Non-atomic Networks. We fix a finite non-empty set $\mathcal{G}$ of *global values*. A *read-write alphabet* is any set of the form $A \times \mathcal{G}$, where A is a set of *read* and *write actions*, or just *reads* and *writes*. We denote a letter $(a, g) \in A \times \mathcal{G}$ by $a(g)$, and write $\mathcal{G}(a_1, \ldots, a_n)$ instead of $\{a_1, \ldots, a_n\} \times \mathcal{G}$.

In what follows, we consider LTSs over read-write alphabets. We fix two LTSs $\mathcal{D}$ and C, called the *leader* and the *contributor*, with alphabets $\mathcal{G}(r_d, w_d)$ and $\mathcal{G}(r_c, w_c)$, respectively, where r_d, r_c are called reads and w_c, w_d are called writes. We write $w_\star$ (respectively, $r_\star$) to stand for either w_c or w_d (respectively, r_c or r_d). We also assume that for each value $g \in \mathcal{G}$ there is a transition in the leader or contributor which reads or writes g (if not, the value is never used and is removed from $\mathcal{G}$).

Additionally, we fix an LTS $\mathcal{S}$ called a *store*, whose states are the global values of $\mathcal{G}$ and whose transitions, labeled with the read-write alphabet, represent possible changes to the global values on reads and writes. No read is enabled initially. Formally, the *store* is an LTS $\mathcal{S} = (\Sigma, \mathcal{G} \cup \{g_0\}, \delta_\mathcal{S}, g_0)$, where $\Sigma = \mathcal{G}(r_d, w_d, r_c, w_c)$, g_0 is a designated initial value not in $\mathcal{G}$, and $\delta_\mathcal{S}$ is the set of transitions $g \xrightarrow{r_\star(g)} g$ and $g' \xrightarrow{w_\star(g)} g$ for all $g \in \mathcal{G}$ and all $g' \in \mathcal{G} \cup \{g_0\}$. Observe that fixing $\mathcal{D}$ and C also fixes $\mathcal{S}$.

Definition 1. *Given a pair $(\mathcal{D}, C)$ of a leader $\mathcal{D}$ and contributor C, and $k \geq 1$, define $\mathcal{N}_k$ to be the LTS $\mathcal{D} \parallel \mathcal{S} \parallel \sqcup_k C$, where $\sqcup_k C$ is $\sqcup_{i=1}^{k} C$. The* (non-atomic) $(\mathcal{D}, C)$-network *$\mathcal{N}$ is the set $\{\mathcal{N}_k \mid k \geq 1\}$, with* language $L(\mathcal{N}) = \bigcup_{k=1}^{\infty} L(\mathcal{N}_k)$. *We omit the prefix $(\mathcal{D}, C)$ when it is clear from the context.*

Notice that $L(\mathcal{N}_k) = L(\mathcal{D}) \parallel L(\mathcal{S}) \parallel \sqcup_k L(C)$ and $L(\mathcal{N}) = L(\mathcal{D}) \parallel L(\mathcal{S}) \parallel \sqcup_\infty L(C)$, where $\sqcup_\infty L(C)$ is given by $\bigcup_{k=1}^{\infty} \sqcup_k L(C)$.

The Safety Verification Problem. A trace of a $(\mathcal{D}, C)$-network $\mathcal{N}$ is unsafe if it ends with an occurrence of $w_c(\#)$, where # is a special value of $\mathcal{G}$. Intuitively, an occurrence of $w_c(\#)$ models that the contributor raises a flag because some error has occurred. A $(\mathcal{D}, C)$-network $\mathcal{N}$ is *safe* iff its language contains no unsafe trace, namely $L(\mathcal{N}) \cap \Sigma^* w_c(\#) = \emptyset$. (We could also require the leader to write #, or to reach a certain state; all these conditions are easily shown equivalent.)

Given a machine M having an LTS semantics over some read-write alphabet, we denote its LTS by $[\![M]\!]$. Given machines M_D and M_C over read-write alphabets, The *safety verification problem* for machines M_D and M_C consists of deciding if the $([\![M_D]\!], [\![M_C]\!])$-network is safe. Notice that the size of the input is the size of the machines, and not the size of the LTSs thereof, which might even be infinite.

Our goal is to characterize the complexity of the safety verification problem considering various types of machines for the leader and the contributors. We first establish some fundamental combinatorial properties of non-atomic networks.

3 Simulation and Monotonicity

We prove two fundamental combinatorial properties of non-atomic networks: the Simulation and Monotonicity Lemmas. Informally, the Simulation Lemma states that a leader cannot distinguish an unbounded number of contributors from the parallel composition of at most $|\mathcal{G}|$ *simulators*—LTSs derivable from the contributors, one for each value

of $\mathcal{G}$. The Monotonicity Lemma states that non-minimal traces (with respect to a certain subword order) can be removed from a simulator without the leader "noticing", and, symmetrically, non-maximal traces can be removed from the leader without the simulators "noticing".

3.1 Simulation

First Writes and Useless Writes. Let σ be a trace. The *first write* of g in σ by a contributor is the first occurrence of $w_c(g)$ in σ. A *useless write* of g by a contributor is any occurrence of $w_c(g)$ that is immediately overwritten by another write. For technical reasons, we additionally assume that useless writes are not first writes.

Example 1. In a network trace $w_d(g_1)_1\, w_c(g_2)_2\, w_c(g_3)_3\ r_d(g_3)_4\, w_c(g_2)_5\, w_c(g_1)_6$ where we have numbered occurrences, $w_c(g_2)_2$ is a first write of g_2, and $w_c(g_2)_5$ is a useless write of g_2 (even though $w_c(g_2)_2$ is immediately overwritten).

We make first writes and useless writes explicit by adding two new actions f_c and u_c to our LTSs, and adequately adapting the store.

Definition 2. *The* extension *of an LTS* $\mathcal{T} = (\mathcal{G}(r,w), Q, \delta, q_0)$ *is the LTS* $\mathcal{T}^E = (\mathcal{G}(r,w,f,u), Q, \delta^E, q_0)$*, where* f, u *are the* first write *and* useless write *actions, respectively, and*

$$\delta^E = \delta \cup \{(q, f(g), q'), (q, u(g), q') \mid (q, w(g), q') \in \delta\} \ .$$

We define an *extended store*, whose states are triples (g, W, b), where $g \in \mathcal{G}$, $W\colon \mathcal{G} \to \{0,1\}$ is the *write record*, and $b \in \{0,1\}$ is the *useless flag*. Intuitively, W records the values written by the contributors so far. If $W(g) = 0$, then a write to g must be a first write, and otherwise a regular write or a useless write. The useless flag is set to 1 by a useless write, and to 0 by other writes. When set to 1, the flag prevents the occurrence of a read. The flag only ensures that between a useless write and the following write no read happens, i.e., that a write tagged as useless will indeed be semantically useless. A regular or first write may be semantically useless or not.

Definition 3. *The* extended store *is the LTS* $S^E = (\Sigma_E, \mathcal{G}_E, \delta_{S^E}, c_0)$ *where*

- $\Sigma_E = \mathcal{G}(r_d, w_d, r_c, w_c, f_c, u_c)$*;*
- $\mathcal{G}_E$ *is the set of triples* (g, W, b)*, where* $g \in \mathcal{G} \cup \{g_0\}$*,* $W : \mathcal{G} \to \{0,1\}$*, and* $b \in \{0,1\}$*;*
- c_0 *is the triple* $(g_0, W_0, 0)$*, where* $W_0(g) = 0$ *for every* $g \in \mathcal{G}$*;*
- δ_{S^E} *has a transition* $(g, W, b) \xrightarrow{a} (g', W', b')$ *where* $g' \in \mathcal{G}$ *iff one of the following conditions hold:*
  - $a = r_\star(g)$*,* $g' = g$*,* $W' = W$*, and* $b = b' = 0$*;*
  - $a = w_d(g')$*,* $W' = W$ *and* $b' = 0$*;*
  - $a = f_c(g')$*,* $W(g') = 0$*,* $W' = W[W(g')/1]$*, and* $b' = 0$*;*
  - $a = w_c(g')$*,* $W(g') = 1$*,* $W' = W$*, and* $b' = 0$*;*
  - $a = u_c(g')$*,* $W(g') = 1$*,* $W' = W$*, and* $b' = 1$.

The extension of $\mathcal{N}_k$ *is* $\mathcal{N}_k^E = \mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_k C^E$ *and the extension of* $\mathcal{N}$ *is the set* $\mathcal{N}^E = \{\mathcal{N}_k^E \mid k \geq 1\}$*. The languages* $L(\mathcal{N}_k^E)$ *and* $L(\mathcal{N}^E)$ *are defined as in Def. 1.*

It follows immediately from this definition that if $v \in L(\mathcal{N}^E)$ then the sequence v' obtained of replacing every occurrence of $f_c(g), u_c(g)$ in v by $w_c(g)$ belongs to $L(\mathcal{N})$. Conversely, every trace v' of $L(\mathcal{N})$ can be transformed into a trace v of $L(\mathcal{N}^E)$ by adequately replacing some occurrences of $w_c(g)$ by $f_c(g)$ or $u_c(g)$.

In the sequel, we use sequences of first writes to partition sets of traces. Define Υ to be the (finite) set of sequences over $\mathcal{G}(f_c)$ with no repetitions. By the very idea of "first writes" no sequence of Υ writes the same value twice, hence no word in Υ is longer than $|\mathcal{G}|$. Also define $\Upsilon_\#$ to be those words of Υ which ends with $f_c(\#)$. Given $\tau \in \Upsilon$, define P_τ to be the language given by $(\Sigma_E \setminus \mathcal{G}(f_c))^* \sqcup\!\sqcup \tau$. P_τ contains all the sequences over Σ_E in which the subsequence of first writes is exactly τ. For $S \subseteq \Upsilon$, $P_S = \bigcup_{\sigma \in S} P_\sigma$.

The Copycat Lemma. Intuitively, a *copycat* contributor is one that follows another contributor in all its actions: it reads or writes the same value immediately after the other reads or writes. Informally, the copycat lemma states that any trace of a non-atomic network can be extended with copycat contributors.

Consider first the non-extended case. Clearly, for every trace of $\mathcal{N}_k$ there is a trace of $\mathcal{N}_{k+1}$ in which the leader and the first k contributors behave as before, and the $(k+1)$-th contributor behaves as a copycat of one of the first k contributors, say the i-th: if the i-th contributor executes a read $r_c(g)$, then the $(k+1)$-th contributor executes the same read *immediately after*, and the same for a write.

Example 2. Consider the trace $r_c(g_0)\, w_d(g_1)\, r_c(g_1)\, w_c(g_2)$ of $\mathcal{D} \parallel \mathcal{S} \parallel C$. Then the sequence $r_c(g_0)^2\, w_d(g_1)\, r_c(g_1)^2\, w_c(g_2)^2$ is a trace of $\mathcal{D} \parallel \mathcal{S} \parallel (C \sqcup\!\sqcup C)$.

For the case of extended networks, a similar result holds, but the copycat copies a first write by a regular write: if the i-th contributor executes an action other than $f_c(g)$, the copycat contributor executes the same action immediately after, but if the i-th contributor executes $f_c(g)$, then the copycat executes $w_c(g)$.

Definition 4. *We say $u \in \mathcal{G}(r_d, w_d)^*$ is* compatible *with a multiset $M = \{v_1, \ldots, v_k\}$ of words over $\mathcal{G}(f_c, w_c, u_c, r_c)$ (possibly containing multiple copies of a word) iff $u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i \neq \emptyset$. Let $\tau \in \Upsilon$. We say u is compatible with M following τ iff $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i) \neq \emptyset$.*

Lemma 1. *Let $u \in \mathcal{G}(r_d, w_d)^*$ and let M be a multiset of words over $\mathcal{G}(r_c, f_c, w_c, u_c)$. If u is compatible with M, then u is compatible with every M' obtained by erasing symbols from $\mathcal{G}(r_c)$ and $\mathcal{G}(u_c)$ from the words of M.*

Proof. Erasing reads and useless writes (that no one reads) by contributors does not affect the sequence of values written to the store and read by someone, hence compatibility is preserved. □

Lemma 2 (Copycat Lemma). *Let $u \in \mathcal{G}(r_d, w_d)^*$, let M be a multiset over $L(C^E)$ and let $v' \in M$. Given a prefix v of v' we have that if u is compatible with M, then u is compatible with $M \oplus v[f_c(g)/w_c(g)]$.*[2]

[2] Throughout the paper, we use $\{\}, \oplus, \ominus$, and $\geq$ for the multiset constructor, union, difference and inclusion, respectively. The word $w[a/b]$ results from w by replacing all occurrences of a by b.

Example 3. $r_d(g_1)$ is compatible with $f_c(g_1)f_c(g_2)$. By the Copycat Lemma $r_d(g_1)$ is also compatible with $\{f_c(g_1)f_c(g_2), w_c(g_1)\,w_c(g_2)\}$. Indeed, $f_c(g_1)\,w_c(g_1)\,r_d(g_1)f_c(g_2)\,w_c(g_2) \in L(S^E)$ is a trace (even though $f_c(g_2)$ is useless).

The Simulation Lemma. The simulation lemma states that we can replace unboundedly many contributors by a finite number of LTSs that "simulate" them. In particular the network is safe iff its simulation is safe.

Let $v \in L(\mathcal{C}^E)$. Let $\#v$ be the number of times that actions of $\mathcal{G}(f_c, w_c)$ occur in v, minus one if the last action of v belongs to $\mathcal{G}(f_c, w_c)$. E.g., $\#v = 1$ for $v = f_c(g_1)r_c(g_1)$ but $\#v = 0$ for $v = r_c(g_1)f_c(g_1)$. The next lemma is at the core of the simulation theorem.

Lemma 3. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(\mathcal{C}^E)$ compatible with u. Then u is compatible with a multiset $\tilde{M}$ over $L(\mathcal{C}^E) \cap \mathcal{G}(r_c, u_c)^*\,\mathcal{G}(f_c, w_c)$.*

Proof. Since u is compatible with M, $u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k} v_i \neq \emptyset$. Lemma 1 shows that we can drop from M all the v_i such that $v_i \in \mathcal{G}(r_c, u_c)^*$. Further, define $\#M = \sum_{i=1}^{k} \#v_i$. We proceed by induction on $\#M$. If $\#M = 0$, then all the words of M belong to $\mathcal{G}(r_c, u_c)^*\ \mathcal{G}(f_c, w_c)$, and we are done. If $\#M > 0$, then there is $v_i \in M$ such that $v_i = \alpha_i\,\sigma\,\beta_i$, where $\alpha_i \in \mathcal{G}(r_c, u_c)^*$, $\sigma \in \mathcal{G}(f_c, w_c)$, and $\beta_i \neq \varepsilon$. Let g be the value written by σ, and let $v_{k+1} = \alpha_i w_c(g)$. By Lemma 2, u is compatible with $\{v_1, \ldots, v_{k+1}\}$, and so there is $v' \in u \parallel L(S^E) \parallel \sqcup\!\sqcup_{i=1}^{k+1} v_i$ in which the write σ of v_i occurs in v' immediately before the write of v_{k+1}. We now let the writes occur in the reverse order, which amounts to replacing v_i by $v_i' = \alpha_i\,u_c(g)\,\beta_i$ and v_{k+1} by $v_{k+1}' = \alpha_i\,\sigma$. This yields a new multiset $M' = M \ominus \{v_i\} \oplus \{v_i', v_{k+1}'\}$ compatible with u. Since $\#M' = \#M - 1$, we then apply the induction hypothesis to M', obtain $\tilde{M}$ and we are done. □

Definition 5. *For all $g \in \mathcal{G}$, let $L_g = L(\mathcal{C}^E) \cap \mathcal{G}(r_c, u_c)^* f_c(g)$. Define S_g be an LTS over $\mathcal{G}(r_c, u_c, f_c, w_c)$ such that $L(S_g) = Pref(L_g \cdot w_c(g)^*)$. Define the LTS $\mathcal{N}^S = \mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_{g\in\mathcal{G}} S_g$ which we call the* simulation *of $\mathcal{N}^E$.*

Lemma 4. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(\mathcal{C}^E) \cap \mathcal{G}(r_c, u_c)^*\ \mathcal{G}(f_c, w_c)$ compatible with u. Then u is compatible with a set $S = \{s_g\}_{g\in\mathcal{G}}$ where $s_g \in L(S_g)$.*

Proof. Let us partition the multiset M as $\{M_g\}_{g\in\mathcal{G}}$ such that M_g contains exactly the traces of M ending with $f_c(g)$ or $w_c(g)$. Note that some M_g might be empty. Each non-empty M_g is of the form $M_g = \{x_1 f_c(g), x_2 w_c(g), \ldots, x_n w_c(g)\}$ where $n \geq 1$, and $x_i \in \mathcal{G}(r_c, u_c)^*$ for every $1 \leq i \leq n$. Define M_g' as empty if M_g is empty, and M_g' as M_g together with $n-1$ copies of $x_1 w_c(g)$. The copycat lemma shows that u is compatible with $\oplus_{g\in\mathcal{G}} M_g'$. Let us now define the multiset M_g'' to be empty if M_g' is empty, and the multiset of exactly n elements given by $x_1 f_c(g)$ and $n-1$ copies of $x_1 w_c(g)$ if M_g' is not empty. Again we show that u is compatible with $\oplus_{g\in\mathcal{G}} M_g''$. The reason is that the number $n-1$ of actions $w_c(g)$ in each M_g'' does not change (compared to M_g) and each $w_c(g)$ action can happen as soon as $f_c(g)$ has occurred.

Now define S consisting of one trace s_g for each $g \in \mathcal{G}$ such that $s_g = \varepsilon$ if $M_g'' = \emptyset$; and $s_g = x_1 f_c(g)\,w_c(g)^{n-1}$ if M_g'' consists of $x_1 f_c(g)$ and $n-1$ copies of $x_1 w_c(g)$.

We have that u is compatible with S because the number of $f_c(g)$ and $w_c(g)$ actions in M_g'' and s_g does not change and each $w_c(g)$ action can happen as soon as $f_c(g)$ has

occurred. Note that it need not be the case that $s_g \in L(C^E)$. However each $s_g \in L(S_g)$ (recall that each $L(S_g)$ is prefix closed). □

Corollary 1. *Let $u \in L(\mathcal{D})$ and let $M = \{v_1, \ldots, v_k\}$ be a multiset over $L(C^E)$ compatible with u. Then u is compatible with a set $S = \{s_g\}_{g \in \mathcal{G}}$ where $s_g \in L(S_g)$.*

In Lemmas 1,2,3 and 4 and Corollary 1 compatibility is preserved. We can further show that it is preserved following a given sequence of first writes. For example, in Lem. 3 if u is compatible with M following τ then u is compatible with $\tilde{M}$ following τ.

Lemma 5 (Simulation Lemma). *Let $\tau \in \Upsilon$:*

$$L(\mathcal{N}^E) \cap P_\tau \neq \emptyset \quad \textit{iff} \quad L(\mathcal{N}^S) \cap P_\tau \neq \emptyset \ .$$

Proof. ($\Rightarrow$): The hypothesis and the definition of $\mathcal{N}^E$ shows that there is $k \geq 1$ such that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_k L(C^E)) \neq \emptyset$.

Therefore we conclude that there exists $u \in L(\mathcal{D})$ and $M = \{v_1, \ldots, v_k\}$ over $L(C^E)$ such that u is compatible with M following τ. Corollary 1 shows that u is compatible following τ with a set $S = \{s_g\}_{g \in \mathcal{G}}$ where $s_g \in L(S_g)$. Therefore we have $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$, hence that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$ and finally that $P_\tau \cap L(\mathcal{N}^S) \neq \emptyset$.

($\Leftarrow$): The hypothesis and the definition of $\mathcal{N}^S$ shows that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \neq \emptyset$. Hence we find that there exists $u \in L(\mathcal{D})$ and a set $\{x_g\}_{g \in \mathcal{G}}$ where $x_g \in L(S_g)$ such that $P_\tau \cap (u \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} x_g) \neq \emptyset$. The prefix closure of $L(S_g)$ shows that either x_g does not have a first write or $x_g = v_g f_c(g) w_c(g)^{n_g}$ for some $v_g f_c(g) \in L_g$ and $n_g \in \mathbb{N}$. In the former case, that is $x_g \in \mathcal{G}(r_c, u_c)^*$, Lemma 1 shows that discarding the trace does not affect compatibility. Then define the multiset M containing for each remaining trace $x_g = v_g f_c(g) w_c(g)^{n_g}$ the trace $v_g f_c(g)$ and n_g traces $v_g w_c(g)$. M contains no other element. Using a copycat-like argument, it is easy to show M is compatible with u and further that compatibility follows τ. Finally, because $v_g f_c(g) \in L(C^E) \cap \mathcal{G}(r_c, u_c)^* \mathcal{G}(f_c)$ and because C^E is the extension of C we find that every trace of M is also a trace of C^E, hence that there exists $k \geq 1$ such that $P_\tau \cap (L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_k L(C^E)) \neq \emptyset$, and finally that $L(\mathcal{N}^E) \cap P_\tau \neq \emptyset$. □

Let us now prove an equivalent safety condition.

Proposition 1. *A $(\mathcal{D}, C)$-network $\mathcal{N}$ is safe iff $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} = \emptyset$.*

Proof. From the semantics of non-atomic networks, $\mathcal{N}$ is unsafe if and only if $L(\mathcal{N}) \cap (\Sigma^* w_c(\#)) \neq \emptyset$, equivalently, $L(\mathcal{N}^E) \cap (\Sigma_E^* f_c(\#)) \neq \emptyset$ (by definition of extension), which in turn is equivalent to $L(\mathcal{N}^E) \cap P_{\Upsilon_\#} \neq \emptyset$ (by definition of $P_{\Upsilon_\#}$), if and only if $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} \neq \emptyset$ (by the simulation lemma). □

3.2 Monotonicity

Before stating the monotonicity lemma, we need some language-theoretic definitions. For an alphabet Σ, define the *subword ordering* $\preceq \subseteq \Sigma^* \times \Sigma^*$ on words as $u \preceq v$ iff u results from v by deleting some occurrences of symbols. Let $L \subseteq \Sigma^*$, define $S \subseteq L$ to be

- *cover* of L if for every $u \in L$ there is $v \in S$ such that $u \preceq v$;
- *support* of L if for every $u \in L$ there is $v \in S$ such that $v \preceq u$.

Observe that for every $u, v \in S$ such that $u \prec v$: if S is a cover then so is $S \setminus \{u\}$, and if S is a support then so is $S \setminus \{v\}$.

Recall that $\mathcal{N}^S = \mathcal{D} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g$. It is convenient to introduce a fourth, redundant component that does not change $L(\mathcal{N}^S)$, but exhibits important properties of it. Recall that the leader cannot observe the reads of the contributors, and does not read the values introduced by useless writes. We introduce a local copy $S^E_{\mathcal{D}}$ of the store with alphabet $\mathcal{G}(r_d, w_d, f_c, w_c)$ that behaves like S^E for writes and first writes of the contributors, but has neither contributor reads nor useless writes in its alphabet. Formally:

Definition 6. *The* leader store $S^E_{\mathcal{D}}$ *is the LTS* $(\Sigma^E_D, \mathcal{G}^{\mathcal{D}}_E, \delta^E_D, c_0)$,
- $\Sigma^E_D = \mathcal{G}(r_d, w_d, f_c, w_c)$;
- $\mathcal{G}^{\mathcal{D}}_E$ *is the set of pairs* (g, W), *where* $g \in \mathcal{G} \cup \{g_0\}$ *and* $W : \mathcal{G} \to \{0, 1\}$;
- c_0 *is the pair* (g_0, W_0), *where* $W_0(g) = 0$ *for every* $g \in \mathcal{G}$;
- δ^E_D *has a transition* $(g, W) \xrightarrow{a} (g', W')$ *where* $g' \in \mathcal{G}$ *iff one of the following conditions hold: a)* $a{=}w_d(g')$ *and* $W' = W$; *b)* $a{=}r_d(g)$, $g' = g$, *and* $W' = W$; *c)* $a{=}f_c(g')$, $W(g') = 0$, *and* $W' = W[W(g')/1]$; *d)* $a{=}w_c(g')$, $W(g') = 1$, *and* $W' = W$.

It follows easily from this definition that $L(S^E_{\mathcal{D}})$ is the projection of $L(S^E)$ onto Σ^E_D, and so $L(S^E) = L(S^E_{\mathcal{D}}) \parallel L(S^E)$ holds. Now, define $\mathcal{DS} = \mathcal{D} \parallel S^E_{\mathcal{D}}$, we find that:

$$\begin{aligned} L(\mathcal{N}^S) &= L(\mathcal{D} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g) && \text{def. 5} \\ &= L(\mathcal{D}) \parallel L(S^E_{\mathcal{D}}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \\ &= L(\mathcal{D} \parallel S^E_{\mathcal{D}}) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \\ &= L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g \in \mathcal{G}} S_g) && (1) \end{aligned}$$

Lemma 6 (Monotonicity Lemma). *Let* $\tau \in \Upsilon$ *and let* $\hat{L}_\tau$ *be a cover of* $L(\mathcal{DS}) \cap P_\tau$. *For every* $g \in \mathcal{G}$, *let* $\underline{L}_g$ *be a support of* L_g, *and let* $\underline{S}_g$ *be an LTS such that* $L(\underline{S}_g) = Pref(\underline{L}_g \cdot w_c^*(g))$:

$$(L(\mathcal{DS}) \cap P_\tau) \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g) \neq \emptyset \;\; \textit{iff} \;\; \hat{L}_\tau \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(\underline{S}_g) \neq \emptyset \;\; .$$

The proof of the monotonicity lemma breaks down into monotonicity for the contributors (Lemma 7) and for the leader (Lemma 9).

Lemma 7 (Contributor Monotonicity Lemma). *For every* $g \in \mathcal{G}$, *let* $\underline{L}_g$ *be a support of* L_g, *and let* $\underline{S}_g$ *be an LTS such that* $L(\underline{S}_g) = Pref(\underline{L}_g w_c^*(g))$. *Let* $u \in \mathcal{G}(r_d, w_d)^*$ *and* $\tau \in \Upsilon$:

$$(u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(S_g)) \cap P_\tau \neq \emptyset \;\; \textit{iff} \;\; (u \parallel L(S^E) \parallel \sqcup_{g \in \mathcal{G}} L(\underline{S}_g)) \cap P_\tau \neq \emptyset \;\; .$$

Proof. ($\Leftarrow$): It suffices to observe that since $\underline{L}_g \subseteq L_g$ we have $L(\underline{S}_g) \subseteq L(S_g)$ and we are done. ($\Rightarrow$): Since $L_g \subseteq \mathcal{G}(r_c, u_c)^* f_c(g)$ and $\underline{L}_g \subseteq L_g$ we find that for every word $w' \in L_g \setminus \underline{L}_g$ there exists a word $w \in \underline{L}_g$ resulting from w' by erasing symbols in $\mathcal{G}(u_c, r_c)$. Hence, Lemma 1 shows that erasing symbols in $\mathcal{G}(u_c, r_c)$ does not affect compatibility. The proof concludes by observing that compatibility is further preserved for τ, and we are done. □

The leader monotonicity lemma requires the following technical observation.

Lemma 8. *Let $\tau \in \Upsilon$ and $L \subseteq \mathcal{G}(r_c, f_c, w_c, u_c)^*$ satisfying the following condition: if $\alpha f_c(g)\beta_1\beta_2 \in L$, then $\alpha f_c(g)\beta_1 w_c(g)\beta_2 \in L$. For every $v, v' \in P_\tau \cap L(S^E_{\mathcal{D}})$:*

$$\text{if } v \parallel L(S^E) \parallel L \neq \emptyset \qquad \text{and } v' \geq v, \qquad \text{then } v' \parallel L(S^E) \parallel L \neq \emptyset \ .$$

Because $v, v' \in P_\tau \cap L(S^E_{\mathcal{D}})$ over alphabet $\Sigma^E_D = \mathcal{G}(r_d, w_d, f_c, w_c)$ and $v' \geq v$ we find that v can be obtained from v' by erasing factors that are necessarily of the form $w_\star(g)\ r_d(g)^*$ or $r_d(g)$. In particular $v, v' \in P_\tau$ shows that $Proj_{\mathcal{G}(f_c)}(v) = Proj_{\mathcal{G}(f_c)}(v') = \tau$.[3] The proof of Lem. 8 is by induction on the number of those factors.

Lemma 9 (Leader Monotonicity Lemma). *Let $\tau \in \Upsilon$ and $L \subseteq \mathcal{G}(r_c, f_c, w_c, u_c)^*$ satisfying: if $\alpha f_c(g)\beta_1\beta_2 \in L$, then $\alpha f_c(g)\beta_1 w_c(g)\beta_2 \in L$. For every cover $\hat{L}_\tau$ of $P_\tau \cap L(\mathcal{DS})$:*

$$(P_\tau \cap L(\mathcal{DS})) \parallel L(S^E) \parallel L \neq \emptyset \quad \textit{iff} \quad \hat{L}_\tau \parallel L(S^E) \parallel L \neq \emptyset \ .$$

Proof. ($\Leftarrow$): It follows from $\hat{L}_\tau \subseteq (P_\tau \cap L(\mathcal{DS}))$. ($\Rightarrow$): We conclude from the hypothesis that there exists $w \in P_\tau \cap L(\mathcal{DS})$ such that $w \parallel L(S^E) \parallel L \neq \emptyset$. Since $\hat{L}_\tau$ is a cover $P_\tau \cap L(\mathcal{DS})$, we find that there exists $w' \in \hat{L}_\tau$ such that $w' \geq w$ ans $w' \in P_\tau \cap L(\mathcal{DS})$. Finally, $\mathcal{DS} = \mathcal{D} \parallel S^E_{\mathcal{D}}$ shows that $w, w' \in P_\tau \cap L(S^E_{\mathcal{D}})$, hence that $w' \parallel L(S^E) \parallel L \neq \emptyset$ following Lem. 8, and finally that $\hat{L}_\tau \parallel L(S^E) \parallel L \neq \emptyset$ because $w' \in \hat{L}_\tau$. □

4 Complexity of Safety Verification of Non-atomic Networks

Recall that the *safety verification problem* for machines M_D and M_C consists in deciding if the $(\llbracket M_D \rrbracket, \llbracket M_C \rrbracket)$-network is safe. Notice that the size of the input is the size of the machines, and not the size of its LTSs, which might even be infinite. We study the complexity of safety verification for different machine classes.

Given two classes of machines `D`, `C` (like finite-state machines or pushdown machines, see below), we define *the class of* (`D`,`C`)*-networks* as the set $\{(\llbracket D \rrbracket, \llbracket C \rrbracket)\text{-network} \mid D \in \mathtt{D}, C \in \mathtt{C}\}$ and denote by `Safety`(`D`, `C`) the restriction of the safety verification problem to pairs of machines $M_D \in \mathtt{D}$ and $M_C \in \mathtt{C}$. We study the complexity of the problem when leader and contributors are *finite-state machines* (FSM) and *pushdown machines* (PDM).[4] In this paper a FSM is just another name for a finite-state LTS, and the LTS $\llbracket A \rrbracket$ of a FSM A is A, i.e. $\llbracket A \rrbracket = A$. We define the size $|A|$ of a FSM A as the size of its transition relation. A (read/write) *pushdown machine* is a tuple $P = (Q, \mathcal{G}(r, w), \Gamma, \Delta, \gamma_0, q_0)$, where Q is a finite set of *states* including the *initial state* q_0, Γ is a *stack alphabet* that contains the *initial stack symbol* γ_0, and $\Delta \subseteq (Q \times \Gamma) \times (\mathcal{G}(r, w) \cup \{\varepsilon\}) \times (Q \times \Gamma^*)$ is a set of *rules*. A *configuration* of a PDM P is a pair $(q, y) \in Q \times \Gamma^*$. The LTS $\llbracket P \rrbracket$ over $\mathcal{G}(r, w)$ associated to P has $Q \times \Gamma^*$ as states, (q_0, γ_0) as initial state, and a transition $(q, \gamma y) \xrightarrow{a} (q', y'y)$ iff $(q, \gamma, a, q', y') \in \Delta$. Define the size of a rule $(q, \gamma, a, q', y') \in \Delta$ as $|y'| + 5$ and the size $|P|$ of a PDM as the sum of the size of rules in Δ.

[3] $Proj_{\Sigma'}(w)$ returns the projection of w onto alphabet Σ'.

[4] We also define FSA and PDA as the automaton (i.e. language acceptor) counterpart of FSM and PDM, respectively. As expected, definitions are identical except for an additional accepting component given by a subset of states in which the automaton accepts.

Determinism. We show that lower bounds (hardness) for the safety verification problems can be achieved already for *deterministic* machines. An LTS $\mathcal{T}$ over a read-write alphabet is *deterministic* if for every state s and every pair of transitions $s \xrightarrow{a_1} s_1$ and $s \xrightarrow{a_2} s_2$, if $s_1 \neq s_2$ then a_1 and a_2 are reads, and they read different values. Intuitively, for any state of a store $\mathcal{S}$, a deterministic LTS $\mathcal{T}$ can take at most one transition in $\mathcal{S} \parallel \mathcal{T}$. A $(\mathcal{D}, C)$-network is deterministic if $\mathcal{D}$ and C are deterministic LTSs. Given a class $\mathcal{X}$ of machines, we denote by $\mathrm{d}\mathcal{X}$ the subclass of machines M of $\mathcal{X}$ such that $[\![M]\!]$ is a deterministic LTS over the read-write alphabet. Notice that this notion does not coincide with the usual definition of a deterministic automaton.

The observation is that a network with non-deterministic processes can be simulated by deterministic ones while preserving safety; intuitively, the inherent non-determinism of interleaving can simulate non-deterministic choice in the machines.

Lemma 10 (Determinization Lemma). *There is a polynomial-time procedure that takes a pair $(\mathcal{D}, C)$ of LTSs and outputs a pair $(\mathcal{D}', C')$ of deterministic LTSs such that the $(\mathcal{D}, C)$-network is safe iff the $(\mathcal{D}', C')$-network is safe.*

We prove the lemma by eliminating non-determinism as follows. Suppose $\mathcal{D}$ is non-deterministic by having transitions $(q, r_d(g), q')$ and $(q, r_d(g), q'')$. To resolve this non-determinism, we define $\mathcal{D}'$ and C' by modifying $\mathcal{D}$ and C as follows: we add new states q_1, q_2, q_3, q_4 to $\mathcal{D}$ and replace the two transitions $(q, r_d(g), q')$ and $(q, r_d(g), q'')$ by the transitions $(q, r_d(g), q_1)$, $(q_1, w_d(\mathbf{nd}), q_2)$, $(q_2, r_d(0), q_3)$, $(q_3, w_d(g), q')$, $(q_2, r_d(1), q_4)$ and $(q_4, w_d(g), q'')$. Let q_0 be the initial state of C. We add two new states $\hat{q}$ and $\tilde{q}$ to C and the transitions $(q_0, r_c(\mathbf{nd}), \hat{q})(\hat{q}, w_c(0), \tilde{q})(\tilde{q}, w_c(1), q_0)$. Finally, we extend the store to accommodate the new values $\{0, 1, \mathbf{nd}\}$. It follows that $\mathcal{D}'$ has one fewer pair of non-deterministic transitions than $\mathcal{D}$. Similar transformations can eliminate other non-deterministic transitions (e.g., two writes from a state) or non-determinism in C.

4.1 Complexity of Safety Verification for FSMs and PDMs

We characterize the complexity of the safety verification problem of non-atomic networks depending on the nature of the leader and the contributors. We show:

`Safety`(dFSM, dFSM), `Safety`(PDM, FSM)	coNP-complete
`Safety`(dPDM, dPDM), `Safety`(PDM, PDM)	PSPACE-complete

Theorem 1. `Safety`(dFSM, dFSM) *is* coNP-*hard.*

We show hardness by a reduction from 3SAT to the complement of the safety verification problem. Given a 3SAT formula, we design a non-atomic network in which the leader and contributors first execute a protocol that determines an assignment to all variables, and uses subsets of contributors to store this assignment. For a variable x, the leader writes x to the store, inviting proposals for values. On reading x, contributors non-deterministically write either "x is 0" or "x is 1" on the store, possibly over-writing each other. At a future point, the leader reads the store, reading the proposal that was last written, say "x is 0." The leader then writes "commit x is 0" on the store. Every contributor that reads this commitment moves to a state where it returns 0 every time the value of x is asked for. Contributors that do not read this message are stuck and do

not participate further. The commitment to 1 is similar. This protocol ensures that each variable gets assigned a consistent value.

Then, the leader checks that each clause is satisfied by querying the contributors for the values of variables (recall that contributors return consistent values) and checking each clause locally. If all clauses are satisfied, the leader writes a special symbol #. The safety verification problem checks that # is never written, which happens iff the formula is unsatisfiable. Finally, Lemma 10 ensures all processes are deterministic.

Theorem 2. `Safety`(PDM, FSM) *is in* coNP.

Proof. Fix a $(\mathcal{D}, C)$-network $\mathcal{N}$, where P_D is a PDM generating $\mathcal{D} = [\![P_D]\!]$, and C is a FSM. Hence $L(\mathcal{D})$ is a context-free language and $L(C)$ is regular. Prop. 1 and Def. 5 (of $\mathcal{N}^S$) show that the $(\mathcal{D}, C)$-network $\mathcal{N}$ is accepting iff $L(\mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$. Since C is given by a FSM, so is C^E. Further, $L_g = L(C^E) \cap \mathcal{G}(r_d, u_c)^* f_c(g)$ has a support captured by those paths in C^E starting from the initial state and whose label ends by $f_c(g)$ and in which no state is entered more than once. Therefore if C^E has k states then the set of paths starting from the initial state, of length at most $k+1$ and whose label ends with $f_c(g)$ is a support, call it $\underline{L}_g$, of L_g. Next, Lem. 7 shows that deciding $L(\mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$ is equivalent to $L(\mathcal{D} \parallel S^E \parallel \sqcup\!\sqcup_{g\in\mathcal{G}} Pref(\underline{L}_g \cdot w_c(g)^*)) \cap P_{\Upsilon_\#} \neq \emptyset$.

Note that this last check does not directly provide a NP algorithm for non-safety because, due to the write records, S^E is exponentially larger than $|\mathcal{G}|$. So, we proceed by pushing down sequences of first writes and obtain the following equivalent statement: $L(\mathcal{D}) \parallel (L(S^E) \cap P_{\Upsilon_\#}) \parallel (\sqcup\!\sqcup_{g\in\mathcal{G}} L(Pref(\underline{L}_g \cdot w_c(g)^*)) \cap P_{\Upsilon_\#}) \neq \emptyset$.

Now, we get an NP algorithm as follows: (*a*) guess $\tau \in \Upsilon_\#$ (this can be done in time polynomial in $|\mathcal{G}|$); (*b*) construct in polynomial time a FSA A_1 for $L(S^E) \cap P_\tau$ (A_1 results from S^E by keeping the $|\tau|$ write records corresponding to τ); (*c*) for each $g \in \tau$, guess $z_g \in \underline{L}_g$ (the guess can be done in polynomial time); (*d*) guess $z \in (\sqcup\!\sqcup_{g\in\mathcal{G}} z_g) \cap P_\tau$ (this fixes a sequence of reads, useless writes and first writes of the contributors according to τ); (*e*) construct in polynomial time a FSA A_2 such that $L(A_2)$ is the least language containing z and if $\alpha f_c(g)\beta_1\beta_2 \in L(A_2)$ then $\alpha f_c(g)\beta_1 w_c(g)\beta_2 \in L(A_2)$ (intuitively we add selfloops with write actions of $\mathcal{G}(w_c)$ to the FSA accepting z such that $w_c(g)$ occurs provided $f_c(g)$ has previously occurred); (*f*) construct in time polynomial in $|P_D|$ a context-free grammar (CFG) G_D such that $L(G_D) = L(P_D)$; (*g*) construct in polynomial time a CFG G such that $L(G) = L(G_D) \parallel L(A_1) \parallel L(A_2)$ (this can be done in time polynomial in $|G_D| + |A_1| + |A_2|$ as shown in the companion technical report [13]); (*h*) check in polynomial time whether $L(G) \neq \emptyset$. □

The complexity of the problem becomes higher when all the processes are PDMs.

Theorem 3. `Safety`(dPDM, dPDM) *is* PSPACE-*hard.*

PSPACE-hardness is shown by reduction from the acceptance problem of a polynomial-space deterministic Turing machine. The proof is technical. The leader and contributors simulate steps of the Turing machine in rounds. The stack is used to store configurations of the Turing machine. In each round, the leader sends the current configuration of the Turing machine to contributors by writing the configuration one element at a time on to the store and waiting for an acknowledgement from some contributor that the

element was received. The contributors receive the current configuration and store the next configuration on their stacks. In the second part of the round, the contributors send back the configuration to the leader. The leader and contributors use their finite state to make sure all elements of the configuration are sent and received.

Additionally, the leader and the contributors use the stack to count to 2^n steps. If both the leader and some contributor count to 2^n in a computation, the construction ensures that the Turing machine has been correctly simulated for 2^n steps, and the simulation is successful. The counting eliminates bad computation sequences in which contributors conflate the configurations from different steps due to asynchronous reads and writes.

Next we sketch the upper PSPACE bound that uses constructions on approximations of context-free languages. Those are detailed in technical report [13].

Theorem 4. `Safety`(PDM, PDM) *is in* PSPACE.

Proof. Let P_D and P_C be PDMs respectively generating $\mathcal{D} = [\![P_D]\!]$ and $C = [\![P_C]\!]$, hence $L(\mathcal{D})$ and $L(C)$ are context-free languages. Proposition 1 shows that the $(\mathcal{D}, C)$-network $\mathcal{N}$ is accepting iff $L(\mathcal{N}^S) \cap P_{\Upsilon_\#} \neq \emptyset$ iff $L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset$ (by (1)). From the construction of the Simulation Lemma, for each $g \in \mathcal{G}$ the language $L_g = L(C^E) \cap \mathcal{G}(r_d, u_c)^* f_c(g)$ is context-free, and so is $L(S_g)$. Given P_C we compute in polynomial time a PDA P_g such that $L(P_g) = L_g$. Next,

$$
\begin{aligned}
& L(\mathcal{DS} \parallel S^E \parallel \sqcup_{g\in\mathcal{G}} S_g) \cap P_{\Upsilon_\#} \neq \emptyset \\
\text{iff } & (L(\mathcal{DS}) \cap P_{\Upsilon_\#}) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} L(S_g) \neq \emptyset \\
\text{iff } & (L(\mathcal{DS}) \cap P_{\Upsilon_\#}) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(L(P_g) \cdot w_c(g)^*) \neq \emptyset \qquad (2) \\
\text{iff } & (\textstyle\bigcup_{\tau\in\Upsilon_\#} \hat{L}_\tau) \parallel L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(\underline{L(P_g)} \cdot w_c(g)^*) \neq \emptyset \qquad (3)
\end{aligned}
$$

(2) follows from definition of S_g and $L_g = L(P_g)$; (3) follows from Lem. 6 and by letting $\hat{L}_\tau$ and $\underline{L(P_g)}$ be a cover and support of $L(\mathcal{DS}) \cap P_\tau$ and $L(P_g)$, respectively.

Next, for all $g \in \mathcal{G}$ we compute a FSA A_g such that $L(A_g)$ is a support of $L(P_g)$. Our first language-theoretic construction shows that the FSA A_g can be computed in time exponential but space polynomial in $|P_g|$. Then, because $L(S^E)$ is a regular language, we compute in space polynomial in $|P_D| + |P_C|$ a FSA A_C such that $L(A_C) = L(S^E) \parallel \sqcup_{g\in\mathcal{G}} \mathit{Pref}(L(A_g) \cdot w_c(g)^*)$. Hence, by (3) and because of $\Upsilon_\#$ (guessing and checking $\tau \in \Upsilon_\#$ is done in time polynomial in $|\mathcal{G}|$) we find that it suffices to prove $\hat{L}_\tau \parallel L(A_C) \neq \emptyset$ is decidable in space polynomial in $|P_D| + |P_C|$.

To compute a cover $\hat{L}_\tau$ of $L(\mathcal{DS}) \cap P_\tau$, we need results about the k-index approximations of a context-free language [4]. Given a CFG G in CNF and $k \geq 1$, we define the *k-index approximation* of $L(G)$, denoted by $L^{(k)}(G)$, consisting of the words of $L(G)$ having a derivation in which every intermediate word contains at most k occurrences of variables. We further introduce an operator $\bowtie$ which, given G and FSA A, computes in polynomial time a context-free grammar $G \bowtie A$ such that $L(G \bowtie A) = L(G) \parallel L(A)$. We prove the following properties:

1. $L^{(3m)}(G)$ is a cover of $L(G)$, where m is the number of variables of G;
2. for every FSA A and $k \geq 1$, $L^{(k)}(G \bowtie A) = L^{(k)}(G) \parallel L(A)$;
3. $L^{(k)}(G) \neq \emptyset$ on input G, k can be decided in NSPACE($k \log(|G|)$).

Equipped with these results, the proof proceeds as follows. Let G_D be a context-free grammar such that $L(G_D) = L(P_D)$. It is well-known that G_D can be computed in time polynomial in $|P_D|$. Next, given τ, we compute a grammar G_D^τ recognizing $P_\tau \cap L(\mathcal{DS})$ as follows. The definition of $\mathcal{DS}$ shows that $P_\tau \cap L(\mathcal{DS}) = L(\mathcal{D}) \parallel (L(S_{\mathcal{D}}^E) \cap P_\tau)$. We then compute a FSA $S_{\mathcal{D}}^\tau$ such that $L(S_{\mathcal{D}}^\tau) = L(S_{\mathcal{D}}^E) \cap P_\tau$. It can be done in time polynomial in $|P_D| + |P_C|$ because it is a restriction of S^E where write records are totally ordered according to τ and there are exactly $|\tau|$ of them. Therefore we obtain, $P_\tau \cap L(\mathcal{DS}) = L(G_D) \parallel L(S_{\mathcal{D}}^\tau)$ because $L(G_D) = L(\mathcal{D})$. Define G_D^τ as the CFG $G_D \bowtie S_{\mathcal{D}}^\tau$ which can be computed in polynomial time in G_D and $S_{\mathcal{D}}^\tau$, hence in $|P_D| + |P_C|$. Clearly $L(G_D^\tau) = P_\tau \cap L(\mathcal{DS})$. Further, $L^{(k)}(G_D^\tau)$ is a cover of $L(G_D^\tau)$ for some $k \leq p(|P_D|)$, where p is a suitable polynomial.

By item 2, $L^{(k)}(G_D^\tau) \parallel L(A_C) = L^{(k)}(G_D^\tau \bowtie A_C)$, where the grammar $G_D^\tau \bowtie A_C$ can be constructed in exponential time and space polynomial in $|P_D| + |P_C|$. Now we apply a generic result of complexity (see e.g. Lemma 4.17, [3]), slightly adapted: given functions $f_1, f_2 \colon \Sigma^* \to \Sigma^*$ and $g \colon \Sigma^* \times \Sigma^* \to \Sigma^*$ if f_i can be computed by a s_{f_i}-space-bounded Turing machine, and g can be computed by a $s_{g_1}(|x_1|) \cdot s_{g_2}(|x_2|)$-space-bounded Turing machine, then $g(f_1(x), f_2(x))$ can be computed in $\log(|f_1(x)| + |f_2(x)|) + s_{f_1}(|x|) + s_{f_2}(|x|) + s_{g_1}(|f_1(x)|) \cdot s_{g_2}(|f_2(x)|))$ space. We have

- f_1 is the function that computes $G_D^\tau \bowtie A_C$ on input (P_D, P_C), and f_2 is the function that on input P_D computes $3m$, where m is the number of variables of G_D^τ. So the output size of f_1 is exponential in the input size, while it is polynomial for f_2. Moreover, s_{f_i} for $i = 1, 2$ is polynomial.
- g is the function that on input $(G_D^\tau \bowtie A_C, 3m)$ yields 1 if $L^{(3m)}(G_D^\tau \bowtie A_C) \neq \emptyset$, and 0 otherwise, where m is the number of variables of G_D^τ. By (3) s_{g_1} is logarithmic, and s_{g_2} is linear.

Finally, the generic complexity result shows that $g \circ f$ can be computed in space polynomial in $|P_D| + |P_C|$, and we are done. □

We note that our three language-theoretic constructions (the construction of automaton A_g that is a cover of $L(P_g)$ of size at most exponential in $|P_g|$, and results 1, 2, and 3 in the proof above) improve upon previous constructions, and are all required for the optimal upper bound. Hague [17] shows an alternate doubly exponential construction using a result of Ehrenfeucht and Rozenberg. This gave a 2EXPTIME algorithm. Even after using our exponential time construction for A_g, we can only get an EXPTIME algorithm, since the non-emptiness problem for (general) context-free languages is P-complete [18]. Our bounded-index approximation for the cover and the space-efficient emptiness algorithm for bounded-index languages are crucial to the PSPACE upper bound.

4.2 The Bounded Safety Problem

Given $k > 0$, we say that a $(\mathcal{D}, C)$-network is *k-safe* if all traces in which the leader and each contributor make at most k steps are safe; i.e., we put a bound of k steps on the runtime of each contributor, and consider only safety within this bound. Here, a step consists of a read or a write of the shared register. The bound does not limit the total length of traces, because the number of contributors is unbounded. The *bounded safety* problem asks, given $\mathcal{D}$, C, and k written in unary, if the $(\mathcal{D}, C)$-network is k-safe.

Given a class of (D,C)-networks, we define BoundedSafety(D, C) as the restriction of the k-safety problem to pairs of machines $M_D \in \mathtt{D}$ and $M_C \in \mathtt{C}$, where we write k in unary. A closer look to Theorem 1 shows that its proof reduces the satisfiability problem for a formula ϕ to the bounded safety problem for a (D,C)-network and a number k, all of which have polynomial size $|\phi|$. This proves that BoundedSafety(dFSM, dFSM) is coNP-hard. We show that, surprisingly, bounded safety remains coNP-complete for pushdown systems, and, even further, for arbitrary Turing machines. Notice that the problem is already coNP-complete for one single Turing machine.

We sketch the definition of the Turing machine model (TM), which differs slightly from the usual one. Our Turing machines have two kind of transitions: the usual transitions that read and modify the contents of the work tape, and additional transitions with labels in $\mathcal{G}(r, w)$ for communication with the store. The machines are input-free, i.e., the input tape is always initially empty.

Theorem 5. BoundedSafety(TM, TM) *is* coNP*-complete.*

Proof. Co-NP-hardness follows from Theorem 1. To prove BoundedSafety(TM, TM) is in NP we use the simulation lemma. Let M_D, M_C, k be an instance of the problem, where M_D, M_C are Turing machines of sizes n_D, n_C with LTSs $\mathcal{D} = [\![M_D]\!]$ and $C = [\![M_C]\!]$, and let $n_D + n_C = n$. In particular, we can assume $|\mathcal{G}| \leq n$, because we only need to consider actions that appear in M_D and M_C. If the $(\mathcal{D}, C)$-network is not k-safe, then by definition there exist $u \in L(\mathcal{D})$ and a multiset $M = \{v_1, \ldots, v_k\}$ over $L(C^E)$ such that u is compatible with M following some $\tau \in \Upsilon_{\#}$; moreover, all of $u, v_1, \ldots, v_m$ have length at most k. By Cor. 1 and Lem. 1 (showing we can drop traces without a first or regular write), there exists a set $S = \{s_{g_1}, \ldots, s_{g_m}\}$ with $m \leq |\mathcal{G}| \leq n$, where $s_{g_i} \in L_{g_i} \cdot w_c(g_i)^*$, and numbers $i_1, \ldots, i_m$ such that u is compatible with $\{s_{g_1} w_c(g_1)^{i_1}, \ldots, s_{g_m} w_c(g_m)^{i_m}\}$ following τ. Since each of the s_{g_i} is obtained by suitably renaming the actions of a trace, we have $|s_{g_i}| \leq k$. Moreover, since the $w_c(g_j)^{i_j}$ parts provide the writes necessary to execute the reads of the s_g sequences, and there are at most $k \cdot (m+1) \leq k \cdot (n+1)$ of them, the numbers can be chosen so that $i_1, \ldots, i_m \leq O(n \cdot k)$ holds.

We present a nondeterministic polynomial algorithm that decides if the $(\mathcal{D}, C)$-network is k-unsafe. The algorithm guesses $\tau \in \Upsilon_{\#}$ and traces $u, s_{g_1}, \ldots, s_{g_m}$ of length at most k. Since there are at most $n+1$ of those traces, this can be done in polynomial time. Then, the algorithm guesses numbers $i_1, \ldots, i_m$. Since the numbers can be chosen so that $i_1, \ldots, i_m \leq O(n \cdot k)$, this can also be done in polynomial time. Finally, the algorithm guesses an interleaving of $u, s_{g_1} w_c(g_1)^{i_1}, \ldots, s_{g_m} w_c(g_m)^{i_m}$ and checks compatibility following τ. This can be done in $O(n^2 \cdot k)$ time. If the algorithm succeeds, then there is a witness that $(L(\mathcal{D}) \parallel L(S^E) \parallel \sqcup\!\sqcup_{g \in \mathcal{G}} L(S_g)) \cap P_\tau \neq \emptyset$ holds, which shows, by Prop. 1 and Def. 5 (of $\mathcal{N}^S$) that the $(\mathcal{D}, C)$-network is unsafe. □

A TM is poly-time if it takes at most $p(n)$ steps for some polynomial p, where n is the size of (the description of) the machine in some encoding. As a corollary, we get that the safety verification problem when leaders and contributors are poly-time Turing machines is coNP-complete. Note that the coNP upper bound holds even though the LTS corresponding to a poly-time TM is exponentially larger than its encoding.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE Computer Society (1996)
2. Apt, K.R., Kozen, D.C.: Limits for automatic verification of finite-state concurrent systems. Information Processing Letters 22(6), 307–309 (1986)
3. Arora, S., Barak, B.: Computational Complexity–A Modern Approach. CUP (2009)
4. Brainerd, B.: An analog of a theorem about context-free languages. Information and Control 11(56), 561–567 (1967)
5. Clarke, E.M., Talupur, M., Veith, H.: Proving Ptolemy right: The environment abstraction framework for model checking concurrent systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 33–47. Springer, Heidelberg (2008)
6. Delzanno, G.: Constraint-based verification of parameterized cache coherence protocols. Formal Methods in System Design 23(3), 257–301 (2003)
7. Delzanno, G., Raskin, J.-F., Van Begin, L.: Towards the automated verification of multithreaded java programs. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 173–187. Springer, Heidelberg (2002)
8. Delzanno, G., Sangnier, A., Zavattaro, G.: Parameterized verification of ad hoc networks. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 313–327. Springer, Heidelberg (2010)
9. Dimitrova, R., Podelski, A.: Is lazy abstraction a decision procedure for broadcast protocols? In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 98–111. Springer, Heidelberg (2008)
10. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
11. Emerson, E.A., Namjoshi, K.S.: Verification of parameterized bus arbitration protocol. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 452–463. Springer, Heidelberg (1998)
12. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359. IEEE Computer Society (1999)
13. Esparza, J., Ganty, P., Majumdar, R.: Parameterized verification of asynchronous shared-memory systems. CoRR abs/1304.1185 (2013)
14. Finkel, A., Leroux, J.: How to compose Presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
15. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. ACM Trans. Program. Lang. Syst. 6, 1–6 (2012)
16. Guerraoui, R., Ruppert, E.: Anonymous and fault-tolerant shared-memory computing. Distributed Computing 20(3), 165–177 (2007)
17. Hague, M.: Parameterised pushdown systems with non-atomic writes. In: Proc. of FSTTCS 2011. LIPIcs, vol. 13, pp. 457–468. Schloss Dagstuhl (2011)
18. Jones, N.D., Laaser, W.T.: Complete problems for deterministic polynomial time. In: Proc. of STOC 1974, pp. 40–46. ACM (1974)
19. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)
20. Laurendeau, C., Barbeau, M.: Secure anonymous broadcasting in vehicular networks. In: LCN 2007, pp. 661–668. IEEE Computer Society (2007)

21. McMillan, K.L.: Verification of an implementation of tomasulo's algorithm by compositional model checking. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 110–121. Springer, Heidelberg (1998)
22. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
23. Viswanathan, M., Chadha, R.: Deciding branching time properties for asynchronous programs. Theoretical Computer Science 410(42), 4169–4179 (2009)

Partial Orders for Efficient Bounded Model Checking of Concurrent Software*

Jade Alglave[1], Daniel Kroening[2], and Michael Tautschnig[3]

[1] University College London
[2] University of Oxford
[3] Queen Mary, University of London

Abstract. The number of interleavings of a concurrent program makes automatic analysis of such software very hard. Modern multiprocessors' execution models make this problem even harder. Modelling program executions with partial orders rather than interleavings addresses both issues: we obtain an efficient encoding into integer difference logic for bounded model checking that enables first-time formal verification of deployed concurrent systems code. We implemented the encoding in the CBMC tool and present experiments over a wide range of memory models, including SC, Intel x86 and IBM Power. Our experiments include core parts of PostgreSQL, the Linux kernel and the Apache HTTP server.

1 Introduction

Automatic analysis of concurrent programs is a challenge in practice. Hardly any of the very few existing tools for software of this kind will prove safety properties for a thousand lines of code [14]. Most papers name the number of *thread interleavings* of a concurrent program as a reason for the difficulty. This view presupposes an execution model, i.e., *Sequential Consistency* (SC) [25], where an execution is a *total order* (more precisely an interleaving) of the instructions from different threads. This execution model poses at least two problems.

First, the large number of interleavings modelling the executions of a program makes their enumeration intractable. *Context bounded* methods [31,23] (unsound in general) and *partial order reduction* [29,17] can reduce the number of interleavings to consider, but still suffer from limited scalability.

Second, modern multiprocessors (e.g., Intel x86 or IBM Power) serve as a reminder that SC is an inappropriate model. Indeed, the *weak memory models* implemented by these chips allow more behaviours than SC. For example, a processor can commit a write first to a store buffer, then to a cache, and finally to memory. While the write is in transit through buffers and caches, a read can occur before the value is actually available to all processors from the memory.

We address both issues by using *partial orders* to model executions, an established theoretical tradition [30,36,6]. We aim at bug finding and practical verification of concurrent programs [11,23,13] – where partial orders have hardly

* Supported by SRC/2269.002, EPSRC/H017585/1, EU FP7 STREP PINCETTE, ARTEMIS/VeTeSS, and ERC/280053.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 141–157, 2013.

ever been considered. Notable exceptions are [33,34] (but we do not have access to an implementation), forming with [10] the closest related work. We show that the explicit use of partial orders generalises these works to concurrency at large, from SC to weak memory. On the technical side, partial orders permit a drastic reduction of the formula size over the use of total orders. Our experiments confirm that this reduction is desirable, as it *increases scalability* by lowering the memory footprint. Furthermore our experiments show, contrasting folklore belief, that the verification time is *hardly affected by the choice of memory model*.

We emphasise that we study *hardware* memory models as opposed to software ones. We believe that verification of concurrent software is still bound to hardware models. Indeed, concurrent systems software is racy on purpose (see our experiments in Sec. 5). Yet, software memory models either banish or give an undefined semantics to racy programs [27,8]. Thus, to give a semantics to concurrent programs, we lift the hardware models to the software level.

In addition to the immediate support of weak memory models, we find that partial orders permit a *very natural and non-intrusive* extension of bounded model checking (BMC) of software to concurrent programs: SAT- and SMT-based BMC builds a formula that describes the data and control flow of a program. For concurrent programs, we do so for each thread of the program. We *add* a conjunct that describes the concurrent executions of these threads as partial orders. We prove that for any satisfying assignment of this formula there is a valid execution w.r.t. our memory models; and conversely, any valid execution gives rise to a satisfying assignment of the formula. We impose no additional bound on context switches, and SC is merely a particular case of our method.

To experiment with our approach, we implement a *symbolic decision procedure for partial orders* in CBMC [12], enabling BMC of concurrent C programs w.r.t. a given memory model for systems code. For SC, we show the efficiency and competitiveness of our approach on the benchmarks of the TACAS 2013 software verification competition [7]. We furthermore support a wide range of weak memory models, including Intel x86 and IBM Power. To exercise our tool on these models, we prove and disprove safety properties in more than 5800 loop-free *litmus tests* previously used to validate formal models against IBM Power chips [32,26]. Our tool is the first to handle the subtle *store atomicity relaxation* [1] specific to Power and ARM. We furthermore perform first-time verification of *core components of systems software*. We show that mutual exclusion is not violated in a queue mechanism of the Apache HTTP server software. We confirm a bug in the worker synchronisation mechanism in PostgreSQL, and that adding two fences fixes the problem. We show that the Read-Copy-Update (RCU) mechanism of the Linux kernel preserves data consistency of the object it is protecting. For all examples we perform the analysis for SC, Intel x86, as well as IBM Power. For each of the systems examples we either succeed in bug finding (for PostgreSQL) or can soundly limit the number of loop iterations required to show the desired property using BMC (RCU is loop-free and the loop in Apache is stateless).

We provide the sources of our tool, our benchmarks and log files, and a long version [3] of the paper, with proof sketches, at http://www.cprover.org/wmm.

P_0	P_1
(a) `x` ← `1`	(c) `y` ← `1`
(b) `r1` ← `y`	(d) `r2` ← `x`
Allowed? `r1=0`; `r2=0`	

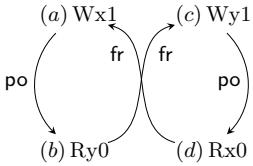


Fig. 1. Store Buffering (**sb**)

P_0	P_1	P_2	P_3
(a) `r1` ← `x`	(c) `r3` ← `y`	(e) `x` ← `1`	(f) `y` ← `1`
(b) `r2` ← `y`	(d) `r4` ← `x`		
Allowed? `r1=1`; `r2=0`; `r3=1`; `r4=0`;			

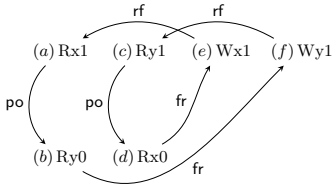


Fig. 2. Independent Reads of Independent Writes (**iriw**)

2 Executions as Partial Orders

Our symbolic decision procedure builds on the framework of [4], which was originally conceived to model weak memory semantics. *Relations* over *read and write memory events* are at the core of this framework.

We introduce this framework on *litmus tests*, as shown in Fig. 1. On the left-hand side we show a multi-threaded program. The shared variables `x` and `y` are initialised to zero. A store (e.g., `x` ← `1` on P_0) gives rise to a write event ((a) Wx1), and a load (e.g., `r1` ← `y` on P_0) to a read event ((b) Ry0). The property of interest is whether there exists an execution of the program such that the final state is `r1=0` and `r2=0`. To determine this, we study the *event graph*, given on the right-hand side of the figure. An architecture allows an execution when it represents an order of all events consistent across all processors. We call this order *global happens before*. A cycle in an event graph is a violation thereof.

For memory models weaker than SC, the architecture possibly *relaxes* some of the relations contributing to this cycle. Such a relaxation makes the graph acyclic, which implies that the architecture allows the final state. In SC, nothing is relaxed, thus the cycle in Fig. 1 forbids the execution. Intel x86 relaxes the program order (po in Fig. 1) between writes and reads, thus the forbidding cycle no longer exists, and the given final state is observed.

Formalisation. An *event* is a read or a write memory access, composed of a unique identifier, a direction (R for read or W for write), a memory address, and a value. We represent each instruction by the events it issues. In Fig. 2, we associate the store `x` ← `1` on processor P_2 with the event (e) Wx1.

The set of events $\mathbb{E}$ and program order, po, form an *event structure*[1] $E \triangleq (\mathbb{E}, \mathsf{po})$; po is a per-processor total order over $\mathbb{E}$. We write $\mathsf{dp} \subseteq \mathsf{po}$ for the relation

[1] We use this term to remain consistent with [4], but note that it differs from G. Winskel's event structures [36].

modelling *dependencies* between instructions, e.g., an *address dependency* occurs when computing an address to access from the value of a preceding load.

We represent the *communication* between processors leading to the final state via an *execution witness* $X \triangleq (\mathsf{ws}, \mathsf{rf})$, which consists of two relations over events. First, the *write serialisation* ws is a per-address total order on writes which models the *memory coherence* widely assumed by modern architectures. It links a write w to any write w' to the same address that hits the memory after w. Second, the *read-from* relation rf links a write w to a read r s.t. r reads the value written by w. We distinguish the internal read-from rfi (between events on the same processor) from the external rfe (between events on distinct processors).

Given a pair of writes $(w_0, w_1) \in \mathsf{ws}$ s.t. $(w_0, r) \in \mathsf{rf}$, we have w_0 globally happening before w_1 by ws and r reading from w_0 by rf. To ensure that r does not read from w_1, we impose that r globally happens before w_1 in the *from-read* relation fr from ws and rf. A read r is in fr with a write w_1 when the write w_0 from which r reads hit the memory before w_1 did. Formally, we have: $(r, w_1) \in \mathsf{fr} \triangleq \exists w_0, (w_0, r) \in \mathsf{rf} \wedge (w_0, w_1) \in \mathsf{ws}$.

In Fig. 2, the final state corresponds to the execution on the right if each memory location initially holds 0. If `r1=1` in the end, the read (a) obtained its value from the write (e) on P_2, hence $(e, a) \in \mathsf{rf}$. If `r2=0` in the end, the read (b) obtained its value from the initial state, thus before the write (f) on P_3, hence $(b, f) \in \mathsf{fr}$. Similarly, we have $(f, c) \in \mathsf{rf}$ from `r3=1`, and $(d, e) \in \mathsf{fr}$ from `r4=0`.

Relaxed or Safe. We model weak memory effects by *relaxing* subrelations of program order or read-from. Thereby [4] provably embraces several models: SC [25], Sun TSO (i.e., x86 [28]), PSO and RMO, Alpha, and a fragment of Power.

We model reads occurring in advance, as described in the introduction, by subrelations of the read-from rf being *relaxed*, i.e., not included in global happens before. When a processor can read from its own store buffer [1] (the typical TSO/x86 scenario), we relax the internal read-from rfi. When two processors P_0 and P_1 can communicate privately via a cache (a case of *write atomicity* relaxation [1]), we relax the external read-from rfe, and call the corresponding write *non-atomic*. This is a particularity of Power or ARM, and cannot happen on TSO/x86. Some program-order pairs may be relaxed by an architecture (defined below) A (e.g., write-read pairs on x86, and all but dp ones on Power), i.e., only a subset of po is guaranteed to occur in this order. This subset is the *preserved program order*, ppo_A. When a relation may not be relaxed, we call it *safe*.

An architecture A may provide *fence* (or *barrier*) instructions to prevent non-SC behaviours. Following [4], the relation $\mathsf{fence}_A \subseteq \mathsf{po}$ induced by a fence is *non-cumulative* when it only orders certain pairs of events surrounding the fence, i.e., fence_A is safe. The relation fence_A is *cumulative* when it makes writes atomic, e.g., by flushing caches. This amounts to making sequences of external read-from and fences ($\mathsf{rfe}; \mathsf{fence}_A$ or $\mathsf{fence}_A; \mathsf{rfe}$) safe, even though rfe alone would not be safe for A. We denote the union of fence_A and additional cumulativity by ab_A.

Architectures. An *architecture* A determines which relations are safe, i.e., embedded in global happens before. Following [4], we always consider the write serialisation ws and the from-read relation fr safe. We denote the safe subset of

read-from, i.e., the read-from relation globally agreed on by all processors, by grf_A. SC relaxes nothing, i.e., rf and po are safe. TSO authorises the reordering of write-read pairs and store buffering but nothing else. Fences are safe by design.

Finally, an execution (E, X) is *valid* on A when three conditions hold: 1. SC holds per address, i.e., communication and program order for accesses with same address $\mathsf{po\text{-}loc}$ are compatible: $\text{uniproc}(E, X) \triangleq \text{acyclic}(\mathsf{ws} \cup \mathsf{rf} \cup \mathsf{fr} \cup \mathsf{po\text{-}loc})$. 2. Values do not come out of thin air, i.e., there is no causal loop: $\text{thin}(E, X) \triangleq \text{acyclic}(\mathsf{rf} \cup \mathsf{dp})$. 3. There exists a linearisation of events in global happens before, i.e., $\mathsf{ghb}_A(E, X) \triangleq \mathsf{ws} \cup \mathsf{grf}_A \cup \mathsf{fr} \cup \mathsf{ppo}_A \cup \mathsf{ab}_A$ does not form a cycle. Formally:

$$\text{valid}_A(E, X) \triangleq \text{uniproc}(E, X) \wedge \text{thin}(E, X) \wedge \text{acyclic}(\mathsf{ghb}_A(E, X))$$

From the validity of executions we deduce a comparison of architectures: We say that an architecture A_2 is *stronger* than another one A_1 when the executions valid on A_2 are valid on A_1. Equivalently we would say that A_1 is *weaker* than A_2. Thus, SC is stronger than any other architecture discussed above.

3 Symbolic Event Structures

For an architecture A and *one* execution witness X, the framework of Sec. 2 determines whether X is valid on A. To prove safety properties of programs, however, we need to reason about all possible executions of the program. To do so efficiently, we use symbolic representations capturing all possible executions in a single constraint system. We then apply SAT or SMT solvers to decide whether a valid execution exists for A, and, if so, get a satisfying assignment corresponding to an execution witness. If no such satisfying assignment exists, the program is proved safe for the given loop unwinding depth.

As said in Sec. 1, we build two conjuncts. The first one, ssa, represents the data and control flow per thread. The second, pord, captures the communications between threads (cf. Sec. 4). We include a reachability property in ssa; the program has a valid execution violating the property iff $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable.

We mostly use *static single assignment form* (SSA) of the input program to build ssa (cf. [21] for details), as common in symbolic execution and bounded model checking. In our SSA variant, each equation is augmented with a *guard*: the guard is the disjunction over all conjunctions of branching guards on paths to the assignment. To counter exponential-sized guards, control-flow join points result in

main	P_0	P_1	P_2	P_3
$x_0 = 0$				
$\wedge\ y_0 = 0$	$\wedge\ r1_0^1 = x_1$	$\wedge\ r3_0^2 = y_2$	$\wedge\ x_3 = 1$	$\wedge\ y_3 = 1$
	$\wedge\ r2_0^1 = y_1$	$\wedge\ r4_0^2 = x_2$		
$\wedge\ \mathsf{prop}$				
$(i_0)\,\mathrm{W}x x_0$				
$(i_1)\,\mathrm{W}y y_0$				
	$(a)\,\mathrm{R}x x_1$	$(c)\,\mathrm{R}y y_2$	$(e)\,\mathrm{W}x x_3$	$(f)\,\mathrm{W}y y_3$
	$(b)\,\mathrm{R}y y_1$	$(d)\,\mathrm{R}x x_2$		

Fig. 3. The formula ssa for **iriw** (Fig. 2) with $\mathsf{prop} = (r1_0^1 = 1 \wedge r2_0^1 = 0 \wedge r3_0^2 = 1 \wedge r4_0^2 = 0)$, and its ses (guards omitted since all true)

simplified guards. To deal with concurrency, we use *a fresh index for each occurrence of a given shared memory variable*, resulting in a fresh symbol in the formula. We add additional equality constraints (cf. Sec. 4.2) to pord to subsequently constrain these. CheckFence [10] and [33,34] use a similar encoding.

Together with ssa, we build a *symbolic event structure* (ses), which captures program information needed to build the second conjunct pord in Sec. 4. Fig. 3 illustrates this section: the formula ssa on top corresponds to the ses beneath.

The top of Fig. 3 gives ssa for Fig. 2. We print a column per thread, vertically following the control flow, but it forms a single conjunction. Each program variable carries its SSA index as a subscript. Each occurrence of the shared memory variables x and y has a unique SSA index. Here we omit the guards, as this program neither uses branching nor loops.

From SSA to Symbolic Event Structures. A symbolic event structure (ses) $\gamma \triangleq (\mathbb{S}, \mathsf{po})$ is a set $\mathbb{S}$ of *symbolic events* and a *symbolic program order* po. A symbolic event holds a *symbolic value* instead of a concrete one as in Sec. 2. We define $\mathrm{g}(e)$ to be the Boolean guard of a symbolic event e, which corresponds to the guard of the SSA equation as introduced above. We use these guards to build the executions of Sec. 2: a guard evaluates to true if the branch is taken, false otherwise. The symbolic program order $\mathsf{po}(\gamma)$ gives a *list of symbolic events per thread* of the program. The order of two events in $\mathsf{po}(\gamma)$ gives the program order in a concrete execution if both guards are true.

We build the ses γ alongside the SSA form, as follows. Each occurrence of a shared program variable on the right-hand side of an assignment becomes a *symbolic read*, with the SSA-indexed variable as symbolic value, and the guard is taken from the SSA equation. Similarly, each occurrence of a shared program variable on the left-hand side becomes a *symbolic write*. Fences do not affect memory states in a sequential setting, hence do not appear in SSA equations. We simply add a fence event to the ses when we see a fence. We take the order of assignments per thread as program order, and mark thread spawn points.

At the bottom of Fig. 3, we give the ses of **iriw**. Each column represents the symbolic program order, per thread. We use the same notation as for the events of Sec. 2, but values are SSA symbols. Guards are omitted again. We depict the thread spawn events by starting the program order in the appropriate row.

From Symbolic to Concrete Event Structures. To relate to the models of Sec. 2, we *concretise* symbolic events. A model V of $\mathsf{ssa} \wedge \mathsf{pord}$, as computed by a satisfiability solver, induces, for each symbolic event, a concrete value (if it is a read or a write) and a valuation of its guard (for both accesses and fences).

The concretisation of a set $\mathbb{S}$ of symbolic events is a set $\mathbb{E}$ of concrete events, as in Sec. 2, s.t. for each $e \in \mathbb{E}$ there is a symbolic version e_s in $\mathbb{S}$. We concretise a symbolic relation similarly. Given an ses γ, $\mathrm{conc}(\gamma, \mathsf{V})$ is the event structure whose set of events is the concretisation of the events of γ w.r.t. V, and whose program order is the concretisation of $\mathsf{po}(\gamma)$ w.r.t. V. For example, the graph of Fig. 2 (erasing the rf and fr relations) concretises the ses of **iriw** (cf. Fig. 3).

4 Symbolic Decision Procedure for Partial Orders

For an architecture A and an $\mathsf{ses}\ \gamma$, we need to represent the communications (i.e., rf, ws and fr) and the weak memory relations (i.e., $\mathsf{ppo}_A, \mathsf{grf}_A$ and ab_A) of Sec. 2. We encode them as a formula pord s.t. $\mathsf{ssa} \wedge \mathsf{pord}$ is satisfiable iff there is an execution valid on A violating the property encoded in ssa. We first describe how we encode partial orders in general, and then discuss the construction and optimisations for each of the above partial orders: the key challenge is to avoid transitive closures in order to obtain a small number of constraints.

4.1 Symbolic Representation of Partial Orders

We associate each symbolic event x of an $\mathsf{ses}\ \gamma$ with a unique *clock* variable clock_x (cf. [24,33]) ranging over the naturals. For two events x and y, we define the Boolean *clock constraint* as $c_{xy} \triangleq (\mathrm{g}(x) \wedge \mathrm{g}(y)) \Rightarrow \mathsf{clock}_x < \mathsf{clock}_y$ ("$<$" being the usual order on natural numbers). We encode a relation r over the symbolic events of γ as the formula $\phi(\mathsf{r})$ defined as the conjunction of the clock constraints c_{xy} for all $(x, y) \in \mathsf{r}$, i.e., $\phi(\mathsf{r}) \triangleq \bigwedge_{(x,y)\in \mathsf{r}} c_{xy}$. The formula $\phi(\mathsf{r}_1 \cup \mathsf{r}_2)$ is equivalent to $\phi(\mathsf{r}_1) \wedge \phi(\mathsf{r}_2)$. Thus we encode unions of relations (e.g., ghb_A) as the conjunction of their respective encodings.

Let C be a valuation of the clocks of γ. Let V be a valuation of the formula ssa associated to γ. One can show that (C, V) satisfies $\phi(\mathsf{r})$ iff the concretisation of r w.r.t. V is acyclic, provided that this relation has finite prefixes.

Overview. We first present our approach on **iriw** (Fig. 2) and its $\mathsf{ses}\ \gamma$ (Fig. 3), and give the construction of constraints for this example. The algorithms implementing the general case for each of the relations are presented in the extended version of this paper [3], which also includes proofs of correctness for each of the algorithms.

(ppo main)	$c_{i_0 i_1}$ (ppo P_0) c_{ab} (ppo P_1) c_{cd}
(rf-val x)	$(s_{i_0 a} \Rightarrow x_1 = x_0) \wedge (s_{i_0 d} \Rightarrow x_2 = x_0) \wedge$ $(s_{ea} \Rightarrow x_1 = x_3) \wedge (s_{ed} \Rightarrow x_2 = x_3)$
(rf-grf x)	$(s_{i_0 a} \Rightarrow c_{i_0 a}) \wedge (s_{ea} \Rightarrow c_{ea}) \wedge$ $(s_{i_0 d} \Rightarrow c_{i_0 d}) \wedge (s_{ed} \Rightarrow c_{ed})$
(rf-some x)	$(s_{i_0 a} \vee s_{ea}) \wedge (s_{i_0 d} \vee s_{ed})$
(ws x)	$\neg c_{i_0 e} \Rightarrow c_{ei_0}$
(fr x)	$((s_{i_0 a} \wedge c_{i_0 e}) \Rightarrow c_{ae}) \wedge ((s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de}) \wedge$ $((s_{ea} \wedge c_{ei_0}) \Rightarrow c_{ai_0}) \wedge ((s_{ed} \wedge c_{ei_0}) \Rightarrow c_{di_0})$

Fig. 4. Partial order constraints for x in Fig. 2 on SC

In Fig. 2, we represent only one possible execution, namely the one corresponding to the (non-SC) final state of the test. In this section, we generate constraints representing all the executions of **iriw** on a given architecture. We give these constraints, for the address x in Fig. 4 in the SC case (for brevity we skip y, analogous to x). As we explain below in detail, weakening the architecture removes some constraints: for example, for Power, we do not include the (rf-grf) and (ppo) constraints. For TSO, all constraints are the same as for SC.

Each symbol c_{ab} of Fig. 4 is a clock constraint, as introduced in Sec. 4.1 above, and thus represents an ordering between the events a and b. A variable s_{wr} represents a read-from between the write w and the read r.

The constraints of Fig. 4 first represent the preserved program order, e.g., on SC or TSO the read-read pairs (a, b) on P_0 (ppo P_0) and (c, d) on P_1 (ppo P_1), but nothing on Power. We generate constraints for the read-from, for example (rf-some x); the first conjunct $s_{i_0 a} \vee s_{ea}$ concerns the read a on P_0. This means that a can read either from the initial write i_0 or from the write e on P_2. The selected read-from pair also implies equalities of the values written and read (rf-val x): for instance, $s_{i_0 a}$ implies that x_1 equals the initialisation x_0. The architecture-independent constraints for write serialisation and from-read are specified as (ws x) and (fr x); (ws y) and (fr y) are analogous. As there are no fences in **iriw**, we do not generate any memory fence constraints.

Valid Executions. We represent the execution of Fig. 2 as follows. For (e, a) and $(i_0, d) \in \mathsf{grf}_A$, we have the constraint $s_{ea} \Rightarrow c_{ea}$ and $s_{i_0 d} \Rightarrow c_{i_0 d}$ in (rf-grf x). This means that a reads from e (as witnessed by s_{ea}), and that we record that e is ordered before a in grf_A (as witnessed by c_{ea}); *idem* for d and i_0. The constraint $(s_{i_0 d} \wedge c_{i_0 e}) \Rightarrow c_{de}$ in (fr x) represents $(d, e) \in \mathsf{fr}$. It reads "if d reads from i_0 and i_0 is ordered before e (in ws, because i_0 and e are two writes to x), then d is ordered before e (in fr)." Together with (ppo P_0) and (ppo P_1), these constraints represent the execution in Fig. 2. We cannot find a satisfying assignment of these constraints, as this leads to both a before b (by (ppo P_0)) and b before a (by (fr y), (rf-grf y), (ppo P_1), (fr x) and (grf x)). On Power, however, we neither have the ppo nor the grf constraints, hence we can find a satisfying assignment.

4.2 Encoding the Axiomatic Memory Model

We now present a systematic account of the encoding of partial orders required for global happens before, as defined in Sec. 2. The constraints for uniproc and thin are only added when not redundant with ghb for a given architecture. When required, their construction follows the same rules as defined for ghb below, but the constraints use distinct sets of clock variables.

Preserved Program Order. As described in Sec. 3, symbolic execution, including loop unrolling, yields lists of symbolic events per thread gathered in $\mathsf{po}(\gamma)$. We encode the required ppo_A via clock constraints derived from $\mathsf{po}(\gamma)$ in the set C_{ppo}. Let $S \in \mathsf{po}(\gamma)$ be a list of events of some thread. We require for any events $e_1, e_2 \in S$ in this order in S, a clock constraint $c_{e_1 e_2}$ to appear in C_{ppo} when:

1. (e_1, e_2) is safe for the architecture A. This test is architecture-specific. For SC, all pairs are safe. IBM Power only guarantees instruction dependencies to be respected, i.e., $c_{e_1 e_2} \in C_{\mathsf{ppo}}$ iff $(e_1, e_2) \in \mathsf{dp}$.
2. There is a control-flow path from e_1 to e_2. This avoids adding clock constraints that are trivially satisfied. Such a case arises when their precondition $\mathrm{g}(e_1) \wedge \mathrm{g}(e_2)$ is false, i.e., the guards cannot both be true.

3. The constraint $c_{e_1 e_2}$ is not in the transitive closure of C_{ppo}. Consider an event e_3 s.t. $c_{e_1 e_2}, c_{e_2 e_3} \in C_{\mathsf{ppo}}$. If all three events share the same guards, i.e., stem from the same control-flow branch, the constraint $c_{e_1 e_3}$ is redundant, as it is in the transitive closure of existing constraints. This is an optimisation.

For the ses γ of **iriw**, we have $\mathsf{po}(\gamma) = \{[i_0, i_1], [a, b], [c, d], [e], [f]\}$. Given an architecture, we thus build the set C_{ppo} as follows: IBM Power only maintains dependencies, which do not exist for the instructions of **iriw**. *Thus C_{ppo} is empty.* RMO relaxes read-read pairs, resulting in $C_{\mathsf{ppo}} = \{c_{i_0 i_1}\}$. For PSO and stronger architectures, read-read pairs are maintained, thus the constraints (ppo P_0) and (ppo P_1) are added as well.

Read-From. We encode read-from (resp. safe read-from) as the set of constraints C_{rf} (resp. C_{grf}). Following Sec. 2, we add constraints to C_{grf} depending on: first, the relation being within one thread or between distinct threads; second, whether A exhibits store buffering, store atomicity relaxation, or both.

The framework of [4] summarised in Sec. 2 follows a post-mortem reasoning with known fixed values, whereas we need to consider all possible executions. Thus, in contrast to Sec. 2, we need to take two additional facts into account:

1. For any read r there are several candidate writes w to the same address. For each such potential pair (w, r) we introduce a free Boolean variable s_{wr}. The set of eligible writes is determined by collecting all writes to the same address as r, with the exception of writes in program order after r (such writes violate the uniproc check of Sec. 2). Each candidate pair contributes a constraint $s_{wr} \Rightarrow c_{wr}$ to C_{grf}, if the pair is safe for the selected architecture A.
   By Sec. 2, rf maps each read to exactly one write. This would induce an exactly-one (pigeon hole) constraint over all s_{wr} for each read. Such constraints can be challenging for CDCL-style SAT solvers; moreover these are redundant in our case: the exclusivity follows from write serialisation and from-read (cf. [3]). We thus instead add a disjunction over all s_{wr} to C_{rf}.
2. For any pair $(w, r) \in \mathsf{rf}$ encoded as s_{wr} we need to ensure that the guard of the write w is true as well as equality over values, because our modified SSA form encoded in ssa has free symbols for each shared memory access. We add such constraints $s_{wr} \Rightarrow \mathrm{g}(w) \wedge x_w = x_r$ to C_{rf}.

For **iriw**, Fig. 4 contains the above encoding in (rf-grf x), (rf-some x) and (rf-val x). For instance, a may read either from the initial write i_0 or from the write e on P_2. These possible pairs are encoded in (rf-grf x), and will be added for all architectures other than Power, which relaxes store atomicity (therefore C_{grf} remains empty on Power). We then enforce that at least one of these read-from pairs exists via the disjunction in (rf-some x). The selected read-from pair also implies equalities of the values written and read (rf-val x): for instance, $s_{i_0 a}$ implies that x_1 equals the initialisation x_0.

Write Serialisation. We encode ws as the set of constraints C_{ws}. By definition, ws is a total order over writes to a given address. We implement the totality by

ensuring that for two writes $w \neq w'$ to the same address either $c_{ww'}$ or $c_{w'w}$ holds, i.e., $\mathsf{clock}_w \neq \mathsf{clock}_{w'}$. Note that if $(w, w') \in \mathsf{po}$, then necessarily $c_{ww'}$ must hold by uniproc. For **iriw** we have $\mathsf{writes} = \{(\mathrm{x}, \{i_0, e\}), (\mathrm{y}, \{i_1, f\})\}$, and the constraint (ws x) for x.

From-Read. We encode fr as the set of constraints C_{fr}. Recall that $(r, w) \in \mathsf{fr}$ means $\exists w'.(w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws}$. We implement the existential quantifier by a disjunction: $\bigvee_{w' \text{is write}} (w', r) \in \mathsf{rf} \wedge (w', w) \in \mathsf{ws} \Rightarrow c_{rw} \in C_{\mathsf{fr}}$. Our implementation expands the disjunction into multiple constraints added to C_{fr}: for each write w' we add a constraint $s_{w'r} \wedge c_{w'w} \Rightarrow c_{rw}$ to C_{fr}. Observe that $s_{w'r}$ encodes $(w', r) \in \mathsf{rf}$, and $c_{w'w}$ encodes $(w', w) \in \mathsf{ws}$.

For **iriw** and x, we obtain the constraint (fr x), where $(s_{i_0 a} \wedge c_{i_0 e}) \Rightarrow c_{ae}$, reads "if $s_{i_0 a}$ is true (i.e., if a reads from i_0), and if $c_{i_0 e}$ is true (i.e., $(i_0, e) \in \mathsf{ws}$) then c_{ae} is true (i.e., a is in fr before e)."

4.3 Memory Fences and Cumulativity

As noted in Sec. 2, to counter the effects of weak memory models, architectures provide fence instructions. We collect their encoding the set C_{ab} which will always be empty on SC. Our implementation supports x86's `mfence` and Power's `sync`, `lwsync` and `isync`. We handle `isync` as part of ppo_A. We first present x86's `mfence` and Power's `sync`, then `lwsync`.

For the non-cumulative part, a fence orders all events in program order before the fence instruction with events in program order after the fence (for `lwsync` this excludes write-read pairs). A naive encoding thereof results in a quadratic number of clock constraints for each fence. For cumulativity, a similar concern applies. To alleviate this, we introduce *fence events*.

Assume fences between the read-read pairs of P_0 and P_1 of **iriw**, and thus fence events s_0 and s_1. We then have $\mathsf{po}(\gamma) = \{[i_0, i_1], [a, s_0, b], [c, s_1, d], [e], [f]\}$. We will instantiate these fences as `sync` and `lwsync` in the following paragraphs and describe the resulting symbolic encodings.

Fences `mfence` and `sync` For all $e, s \in S$ for some $S \in \mathsf{po}(\gamma)$, with s being a fence event, we first build constraints for *non-cumulativity*: if e is before (resp. after) s in program order, we add c_{es} to C_{ab} (resp. c_{se}).

In **iriw** with the additional fences mentioned above instantiated with `sync`, we generate c_{as_0} (resp. c_{cs_1}) for the event a (resp. c) in po before the fence s_0 (resp. s_1) on P_0 (resp. P_1). We generate $c_{s_0 b}$ (resp. $c_{s_1 d}$) for b (resp. d), in po after the fence s_0 (resp. s_1) on P_0 (resp. P_1).

If stores are not atomic, we build *cumulativity constraints*. For A-cumulativity, we add the constraint $s_{we} \Rightarrow c_{ws}$, for each (w, e) s.t. e is in po before the fence s, and e reads from the write w. The constraint reads "if s_{we} is true (i.e., e reads from w), then c_{ws} is true (i.e., there is a global ordering, due to the fence s, from w to s)". All other constraints, i.e., the actual ordering of w before some event e' in po *after* s, follow by transitivity. We handle B-cumulativity in a similar way.

As Power relaxes store atomicity, the `sync` fences between the read-read pairs of **iriw** create A-cumulativity constraints, namely for s_0 (and analogous ones for s_1): $(s_{i_0 a} \Rightarrow c_{i_0 s_0}) \wedge (s_{ea} \Rightarrow c_{es_0})$.

Fence `lwsync`. As `lwsync` does not order write-read pairs (cf. Sec. 2), we need to avoid creating a constraint c_{wr} between a write w and a read r separated by an `lwsync`. To do so, we use two distinct clock variables clock_s^r and clock_s^w for an `lwsync` s. This avoids the wrong transitive constraint c_{wr} implied by c_{ws} and c_{sr}. Fig. 5 illustrates this setup: the write-read pair (w_1, r_2) will not be ordered by any of the constraints, but all other pairs are ordered.

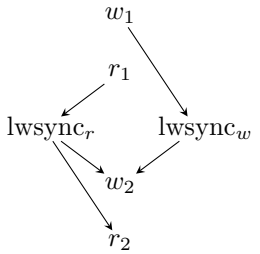


Fig. 5. `lwsync`'s constraints

To create a clock constraint, we then pick one or both clock variables, as follows. If e is a read, the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^r$ when e is before s (or $\mathsf{clock}_s^r < \mathsf{clock}_e$ if e is after). If e is a write preceding s, the clock constraint is $\mathsf{clock}_e < \mathsf{clock}_s^w$. Finally, if e is a write after s, the clock constraint is the conjunction $(\mathsf{clock}_s^w < \mathsf{clock}_e) \wedge (\mathsf{clock}_s^r < \mathsf{clock}_e)$.

In **iriw**, if we use `lwsync` instead of `sync` as discussed above, we obtain the following constraints: $(\mathsf{clock}_a < \mathsf{clock}_{s_0}^r) \wedge (\mathsf{clock}_{s_0}^r < \mathsf{clock}_b) \wedge (s_{i_0 a} \Rightarrow \mathsf{clock}_{i_0} < \mathsf{clock}_{s_0}^w) \wedge (s_{ea} \Rightarrow \mathsf{clock}_e < \mathsf{clock}_{s_0}^w)$. These constraints will *not* order the writes i_0 or e with the read b, because i_0 and e are ordered w.r.t. to $\mathsf{clock}_{s_0}^w$, but b is only ordered w.r.t. the distinct $\mathsf{clock}_{s_0}^r$. This corresponds to the fact that placing `lwsync` fences in **iriw** does not forbid the non-SC execution.

4.4 Soundness and Completeness of the Encoding

Given an architecture A and a program, the procedure of Sec. 3 and Sec. 4 outputs a formula $\mathsf{ssa} \wedge \mathsf{pord}$ and an ses γ. This formula provably encodes the executions of this program valid on A and violating the property encoded in ssa in a sound and complete way. Proving this requires showing that any assignment to the system corresponds to a valid execution of the program, and vice versa. This result requires three steps, one for uniproc, one for thin and one for the acyclicity of ghb. By lack of space, we show only the last one. Given an ses γ, we write ϕ for $\bigwedge_{c \in C_{\mathsf{ppo}} \cup C_{\mathsf{grf}} \cup C_{\mathsf{ws}} \cup C_{\mathsf{fr}} \cup C_{\mathsf{ab}}} c$:

Theorem 1. *The formula $\mathsf{ssa} \wedge \phi$ is satisfiable iff there are V, a valuation of ssa, and a well formed X s.t. $\mathsf{ghb}_A(\mathrm{conc}(\gamma, \mathsf{V}), X)$ is acyclic and has finite prefixes.*

To decide the satisfiability of ϕ, we can use any solver supporting propositional combinations of integer difference logic constraints. The procedure reveals the concrete executions, as expressed by Thm. 1.

5 Experimental Results

We implemented the encoding described above in the bounded model checker CBMC [12], built with the SAT solver MiniSat 2.2.0 as back-end decision procedure. We study the efficiency on standard benchmarks, show the support and

correct implementation of a broad range of memory models on litmus tests, and demonstrate the real-world fitness on widely deployed systems code. The full raw data of our results are available on our web page http://www.cprover.org/wmm.

As is elaborated below, the limited availability of proposed related techniques as well as, where available, unfitness to process real-world C programs, restricts what we can conclude about pre-existing techniques. Yet we find that our technique is scalable enough to verify non-trivial, real-world concurrent systems code, including the worker-synchronisation logic of the relational database PostgreSQL, code for socket-handover in the Apache httpd, and the core API of the Read-Copy-Update (RCU) mutual exclusion code from Linux 3.2.21.

In Tab. 1 we present key facts of our benchmarks: we give the average over all 34 examples of the Software Verification Competition 2013 [7]; similarly we use averages for our 5800 litmus tests; the last three columns provide data for the systems code we study: the worker synchronisation in PostgreSQL, RCU, and fdqueue in Apache httpd. For each we give the number of lines of code (LOC), and the loop unwinding bound used in the experiments (loop unwind) – "none" when there is no loop, and "bounded" when the loops in the program are natively bounded. The number of equations in the resulting ssa is listed as SSA size. We further list key characteristics concerning the symbolic encoding of partial orders: the number of distinct shared memory addresses (#addresses), the total number of shared memory accesses plus fence events (#events), the maximal number of accesses to a single address (max/addr), the total number of constraints required for the partial order encoding (#constraints), and the relation accounting for the largest fraction of constraints (most costly).

Table 1. Statistics about all examples

	SV-COMP	Litmus	PgSQL	RCU	Apache
LOC	798.3	51.2	5412	5834	28864
loop unwind	6	none	1	bounded	5
SSA size	716	80.2	245	161	1027
#addresses	5.8	6.6	5	7	9
#events	160.2	40.9	74	37	140
max/addr	53.6	3.8	17	4	93
#constraints	3576.4	362.0	1089	393	1137
most costly	rf (1587)	rf (81.3)	rf (306)	rf (67)	rf (247)

Observe that the total number of shared accesses is on average more than 5.7 times the maximal number of accesses to a single address, making a strong case for the use of partial orders: *the number of constraints generated for total orders would thus be larger by a factor of* 5^3*, i.e., two orders of magnitude more costly.* The most costly constraint is usually the encoding of read-from.

Other Tools. Few tools verify concurrent C programs, even on SC [14]. In particular, the implementation of [33,34] is not available. For weak memory, solutions were restricted to TSO, and its siblings PSO and RMO [10,5,22], of which only CheckFence [10] is available and able to handle C programs. With [2] we were the first to present a program transformation-based approach for weaker models.

In addition to CheckFence, we tried five further tools, covering a range of techniques for verifying C programs on SC: SatAbs [11], based on predicate abstraction; ESBMC [13], a bounded model checker exploring interleavings with partial order reduction; Threader [19], a thread-modular verifier; CSeq [15] and Poirot, both implementing a context-bounded translation to sequential programs [23]. Poirot and CheckFence, however, could only parse litmus tests.

5.1 Efficiency: SV-COMP'13

We use the 34 concurrency benchmarks from https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/ to compare the efficiency of the partial order based approach to existing tools. We mirror the competition settings, with a time-out of 15 minutes and a memory bound of 15 GB. Fig. 6 (left) depicts the overall performance of SatAbs (7 solved correctly w.r.t. the rules of SV-COMP), CSeq (12), ESBMC (15), Threader (29), and CBMC, which solves all 34 instances correctly. The programs where CBMC takes more than a few seconds have a large number of shared memory array operations, which challenge the underlying SAT solver even for the SSA part. We primarily compare the run-time with ESMBC, as it also performs bounded model checking, but analyses interleavings (total orders). Fig. 6 (right, logarithmically scaled) shows that CBMC outperforms ESBMC on all examples. Comparing to Threader is less meaningful, as Threader is abstraction-based and does not impose loop bounds. We note that Threader wrongly marks one benchmark (qrcu) as safe, whereas CBMC correctly reports a counterexample. No other tool had been able to analyse the program, thus Threader's result had been deemed correct for SV-COMP'13. This was raised with the competition organiser and the developers of Threader.

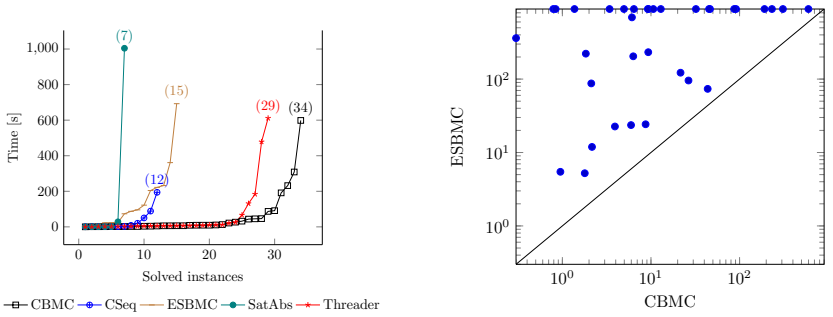


Fig. 6. Comparison of efficiency on SV-COMP'13 benchmarks

5.2 Weak Memory Models: Litmus Tests

We analyse 5803 tests exposing weak memory artefacts, e.g., instruction reordering, store buffering, store atomicity relaxation. These tests are assembly

programs with a non-SC final state, but reachable on a weaker model, generated by the diy tool [4]. For example, **iriw** (Fig. 2) can only be reached on RMO (by reordering the reads) or on Power (*idem*, or because the writes are non-atomic).

We convert these tests into C code, of 51 lines on average, involving 2 to 6 threads. Despite the small size of the tests, they prove challenging to verify: as we showed in [2], most tools, except Blender [22], SatAbs and CBMC, give wrong results or fail in other ways on a vast majority of tests, even for SC, when run for up to 15 minutes. CBMC, however, takes 0.21 s on average to correctly compute the result for each of the memory models SC, TSO, PSO, RMO, Alpha, and Power. No test requires more than 0.7 s, with the exception of the test CO-IRIW, which takes up to 3.7 s (it yields 2450 partial order constraints). CheckFence reported violated properties on all tests, even on SC (where all properties of these tests hold). Blender, which supports only PSO, took 0.6 s on average, and at most 9.7 s. With [2] we can transform C programs to analyse them under weak memory model semantics with SC-only tools. For these transformed programs, SatAbs took 87.8 s on average, Poirot 364.1 s, and ESBMC 723.1 s (all three tools also timed out on several instances). Analysing the transformed programs with CBMC, and SC as memory model, took 6.7 s on average and 305 s at most.

5.3 Real-World Systems Code

We study key components of software widely deployed in server systems. Other tools, including ESBMC and Threader, largely fail to even parse the code.

PostgreSQL. Developers observed that a regression test failed on a PowerPC machine,[2] and later identified the memory model as possible culprit: the processor could delay a write by a thread until after a token signalling the end of this thread's work had been set. A detailed description of the problem is in [2]. Our tool confirms the bug, and proves a patch we proposed to fix the problem. For each memory model, and both with and without the fix, CBMC takes 3 s.

Read-Copy-Update (RCU) is a synchronisation mechanism of the Linux kernel. Writers to a concurrent data structure prepare a fresh component (e.g., list element), then replace the existing component by adjusting the pointer variable linking to it. The old component is cleaned up when there is no process reading.

Thus readers can rely on lightweight (hence fast) lock-free synchronisation. Protection of reads against concurrent writes is fence-free on x86, and uses only a lightweight fence (`lwsync`) on Power. We verify the original implementation of the 3.2.21 kernel for x86 and Power in less than 1 s, using a harness that asserts that the reader will not obtain an inconsistent version of the component. On Power, removing the `lwsync` makes the assertion fail.

Apache httpd is the most widely used HTTP server software. It supports a broad range of concurrency APIs distributing incoming requests to a pool of workers.

[2] http://archives.postgresql.org/pgsql-hackers/2011-08/msg00330.php

The fdqueue module (28864 lines) is the central part of this mechanism, which implements the hand-over of a socket together with a memory pool to an idle worker. The implementation uses a central, shared queue for this purpose. Shared access is synchronised via an integer keeping track of the number of idle workers, which is updated via architecture-dependent compare-and-swap and atomic decrement operations. Hand-over of the socket and the pool and wake-up of the idle thread is then coordinated by means of a conventional, heavy-weight mutex and a signal. We show that hand-over guarantees consistency of the payload data passed to the worker. The architecture-dependent code is only verified by CBMC, in less than 70 seconds.

6 Related Work and Conclusion

We broadly survey verification for concurrent programs in [3]. Here, we focus on closely related methods for software verification, and weak memory models.

Most existing work for weak memory models supports assembly or toy languages only [18,35,26,9], except for [5,20,2] and [10]. Yet [5] bounds the number of context switches, is restricted to TSO, and is not automated. The work of [20] implements an explicit-state analysis for C#. In our prior work [2] we use program transformation to verify C programs w.r.t. weak memory model semantics *using existing SC model checkers*. We discuss [10], which has been successfully applied to non-trivial algorithms, in detail below.

Our work relates the most to [10,16,33,34], which use axiomatic specifications of SC to compose the distinct threads and a similar SSA encoding per thread. The size of the encodings of [10,33,34] are $\mathcal{O}(N^3)$ for N shared memory accesses *to any address*, as we detail below; [16] is quadratic, but in the number of threads times the number of per-thread transitions, which may include arbitrary many local accesses. Our encoding is $\mathcal{O}(M^3)$ (due to fr and ab, others are quadratic only), with M the maximal number of events for a *single address*. In Sec. 5, N on average was 5.7 times larger than M. This extrapolates to a difference of more than *two orders of magnitude* in the size of the formula.

CheckFence [10] encodes total orders over memory accesses. In contrast to our clock variables, [10] uses a Boolean variable M_{xy} per pair (x, y), such that M_{xy} places x and y in a total order: either x before y, or y before x. Furthermore, transitive closure constraints are required; their number is at least cubic in the number of variables M_{xy}. We only consider relations per address, except for program order and fences, and do not build transitive closures. As noted above, the constraints for fr and ab are cubic in the worst case; all others are quadratic.

Sinha and Wang [33,34] use partial orders like us; they note redundancies in their constraints and then develop pruning [33] and abstractions [34] to reduce these. Initially, [33,34] quantify over all events regardless of their address, whereas we mostly build constraints per address, based on our formal framework. As said above, this results in two orders of magnitude lower formula size.

Conclusion. We developed a symbolic encoding of partial orders to perform bounded model checking of concurrent software. We generalise [33,34] to weak

memory, also unsupported by [16]. We prove suitability and scalability of our tool to systems code, which could not be processed by CheckFence; implementations of [16,33,34] are not available. We furthermore showed superior performance on benchmarks of SV-COMP, comparing favourably to all participants.

Acknowledgements. We thank Matthew Hague, Alex Horn, Lihao Liang, Vincent Nimal, Peter O'Hearn and Georg Weissenbacher for invaluable discussions and comments.

References

1. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer (1995)
2. Alglave, J., Kroening, D., Nimal, V., Tautschnig, M.: Software verification for weak memory via program transformation. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 512–532. Springer, Heidelberg (2013)
3. Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient BMC of concurrent software. CoRR abs/1301.1629 (2013)
4. Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in Weak Memory Models (Extended Version). In: FMSD (2012)
5. Atig, M.F., Bouajjani, A., Parlato, G.: Getting rid of store-buffers in the analysis of weak memory models. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 99–115. Springer, Heidelberg (2011)
6. Ben-Asher, Y., Farchi, E.: Using True Concurrency to Model Execution of Parallel Programs. In: IJPP (1994)
7. Beyer, D.: Second competition on software verification. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 594–609. Springer, Heidelberg (2013)
8. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI (2008)
9. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: Felleisen, M., Gardner, P. (eds.) ESOP 2013. LNCS, vol. 7792, pp. 533–553. Springer, Heidelberg (2013)
10. Burckhardt, S., Alur, R., Martin, M.: CheckFence: Checking consistency of concurrent data types on relaxed memory models. In: PLDI (2007)
11. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
12. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
13. Cordeiro, L., Fischer, B.: Verifying multi-threaded software using SMT-based context-bounded model checking. In: ICSE (2011)
14. D'Silva, V., Kroening, D., Weissenbacher, G.: A survey of automated techniques for formal software verification. TCAD (2008)
15. Fischer, B., Inverso, O., Parlato, G.: CSeq: A sequentialization tool for C. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 616–618. Springer, Heidelberg (2013)

16. Ganai, M., Gupta, A.: Efficient modeling of concurrent systems in BMC. In: SPIN. Springer, Heidelberg (2008)
17. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer (1996)
18. Gopalakrishnan, G.C., Yang, Y., Sivaraj, H.: QB or not QB: An efficient execution verification tool for memory orderings. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 401–413. Springer, Heidelberg (2004)
19. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
20. Huynh, Q., Roychoudhury, A.: A memory sensitive checker for C#. In: FM (2006)
21. Kroening, D., Clarke, E., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC (2003)
22. Kuperstein, M., Vechev, M., Yahav, E.: Automatic inference of memory fences. In: FMCAD (2010)
23. Lal, A., Reps, T.: Reducing concurrent analysis under a context bound to sequential analysis. In: FMSD (2009)
24. Lamport, L.: Time, Clocks, and the Ordering of Events in a Distributed System. CACM (1978)
25. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. (1979)
26. Mador-Haim, S., Maranget, L., Sarkar, S., Memarian, K., Alglave, J., Owens, S., Alur, R., Martin, M.M.K., Sewell, P., Williams, D.: An axiomatic memory model for POWER multiprocessors. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 495–512. Springer, Heidelberg (2012)
27. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL (2005)
28. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 391–407. Springer, Heidelberg (2009)
29. Peled, D.: All from one, one for all: on model checking using representatives. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 409–423. Springer, Heidelberg (1993)
30. Pratt, V.: Modeling Concurrency with Partial Orders. International Journal of Parallel Programming (1986)
31. Qadeer, S., Rehof, J.: Context-bounded model checking of concurrent software. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 93–107. Springer, Heidelberg (2005)
32. Sarkar, S., Sewell, P., Alglave, J., Maranget, L., Williams, D.: Understanding Power Multiprocessors. In: PLDI (2011)
33. Sinha, N., Wang, C.: Staged concurrent program analysis. In: FSE (2010)
34. Sinha, N., Wang, C.: On interference abstractions. In: POPL (2011)
35. Torlak, E., Vaziri, M., Dolby, J.: MemSAT: Checking axiomatic specifications of memory models. In: PLDI (2010)
36. Winskel, G.: Event structures. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 325–392. Springer, Heidelberg (1987)

Incremental, Inductive Coverability

Johannes Kloos, Rupak Majumdar, Filip Niksic, and Ruzica Piskac

MPI-SWS, Kaiserslautern and Saarbrücken

Abstract. We give an incremental, inductive (IC3) procedure to check coverability of well-structured transition systems. Our procedure generalizes the IC3 procedure for safety verification that has been successfully applied in finite-state hardware verification to infinite-state well-structured transition systems. We show that our procedure is sound, complete, and terminating for *downward-finite* well-structured transition systems —where each state has a finite number of states below it— a class that contains extensions of Petri nets, broadcast protocols, and lossy channel systems.

We have implemented our algorithm for checking coverability of Petri nets. We describe how the algorithm can be efficiently implemented without the use of SMT solvers. Our experiments on standard Petri net benchmarks show that IC3 is competitive with state-of-the-art implementations for coverability based on symbolic backward analysis or expand-enlarge-and-check algorithms both in time and space usage.

1 Introduction

The IC3 algorithm [3] was recently introduced as an efficient technique for safety verification of hardware. It computes an inductive invariant by maintaining a sequence of over-approximations of forward-reachable states, and incrementally strengthening them based on counter-examples to inductiveness. The counter-examples are obtained using a backward exploration from error states. Efficient implementations of the procedure show remarkably good performance on hardware benchmarks [8].

A natural direction is to extend the IC3 algorithm to classes of systems beyond finite-state hardware circuits. Indeed, an IC3-like technique was recently proposed for interpolation-based software verification [5], and the technique was generalized to finite-data pushdown systems, as well as systems using linear real arithmetic, such as timed pushdown automata [15]. It is natural to ask for what other classes of infinite-state systems does IC3 form a decision procedure for safety verification.

In this paper, we consider well-structured transition systems (WSTS) [1,12]. WSTS are infinite-state transition systems whose states have a well-quasi-ordering, and whose transitions satisfy a monotonicity property w.r.t. the quasi-ordering. WSTS capture many important infinite-state models, such as Petri nets and their monotonic extensions [11,4,7,13], broadcast protocols [9,10], and lossy channel systems [2]. A general decidability result shows that the coverability problem (reachability in an upward-closed set) is decidable for WSTS [1]. The

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 158–173, 2013.

decidability result performs a backward reachability analysis, and shows, using properties of well-quasi-orderings, that the reachability procedure must terminate. In many verification problems, techniques based on computing inductive invariants outperform methods based on backward or forward reachability analysis; indeed, IC3 for hardware circuits is a prime example. Thus, it is natural to ask if there is an IC3-style decision procedure for the coverability problem.

We answer this question positively. We give a generalization of IC3 for WSTS, and show that it terminates on the class of *downward-finite* WSTS, in which each state has a finite number of states lower than itself in the well-quasi-ordering. The class of downward-finite WSTS contains most important classes of WSTS used in verification, including Petri nets and their extensions, broadcast protocols, and lossy channel systems. Hence, our results show that IC3 is a decision procedure for the coverability problem for these classes of systems. While termination is trivial in the finite-state case, our technical contribution is to show, using the termination of the backward reachability procedure, that the sequence of (downward-closed) invariants produced by IC3 is guaranteed to converge. We also show that the assumption of downward-finiteness is necessary: we give an example of a general WSTS on which the algorithm does not terminate.

We have implemented our algorithm in a tool called IIC to check coverability in Petri nets. Using combinatorial properties of Petri nets, we derive an optimized implementation of the algorithm that does not use an SMT solver. Our implementation outperforms, both in space and time usage, several other implementations of coverability, such as EEC [13] or backward reachability, on a set of standard Petri net benchmarks.

A full version, including proofs of theorems, is available on arXiv [17].

2 Preliminaries

Well-quasi-orderings. For a set X, a relation $\preceq \subseteq X \times X$ is a *well-quasi-ordering (wqo)* if it is reflexive, transitive, and if for every infinite sequence $x_0, x_1, \ldots$ of elements from X, there exists $i < j$ such that $x_i \preceq x_j$. A set $Y \subseteq X$ is *upward-closed* if for every $y \in Y$ and $x \in X$, $y \preceq x$ implies $x \in Y$. Similarly, a set $Y \subseteq X$ is *downward-closed* if for every $y \in Y$ and $x \in X$, $x \preceq y$ implies $x \in Y$. For a set Y, we define its upward closure $Y\uparrow = \{x \mid \exists y \in Y, y \preceq x\}$. For a singleton $\{x\}$, we simply write $x\uparrow$ instead of $\{x\}\uparrow$. Similarly, we define $Y\downarrow = \{x \mid \exists y \in Y, x \preceq y\}$ for the downward closure of a set Y. Clearly, $Y\uparrow$ (resp., $Y\downarrow$) is an upward-closed set (resp. downward-closed) for each Y. The union and intersection of upward-closed sets are upward-closed, and the union and intersection of downward-closed sets are downward-closed. Furthermore, the complement of an upward-closed set is downward-closed, and vice versa. For the convenience of the reader, we will mark upward-closed sets with a small up-arrow superscript, like this: $U^{\uparrow}$, and downward-closed sets with a small down-arrow superscript, like this: $D^{\downarrow}$.

A basis of an upward-closed set Y is a set $Y_b \subseteq Y$ such that $Y = \bigcup_{y \in Y_b} y\uparrow$. It is known [14,1,12] that any upward-closed set Y in a wqo has a finite basis: the set of minimal elements of Y has finitely many equivalence classes under

the equivalence relation $\preceq \cap \succeq$, so take any system of representatives. We write $\min Y$ for such a system of representatives. Moreover, it is known that any non-decreasing sequence $I_0 \subseteq I_1 \subseteq \ldots$ of upward-closed sets eventually stabilizes, i.e., there exists $k \in \mathbb{N}$ such that $I_k = I_{k+1} = I_{k+2} = \ldots$.

A wqo $(X, \preceq)$ is *downward-finite* if for each $x \in X$, the downward closure $x{\downarrow}$ is a finite set.

Examples. Let $\mathbb{N}^k$ be the set of k-tuples of natural numbers, and let $\preceq$ be pointwise comparison: $v \preceq v'$ if $v_i \leqslant v'_i$ for $i = 1, \ldots, k$. Then, $(\mathbb{N}^k, \preceq)$ is a downward-finite wqo [6].

Let A be a finite alphabet, and consider the subword ordering $\preceq$ on words over A, given by $w \preceq w'$ for $w, w' \in A^*$ if w results from w' by deleting some occurrences of symbols. Then $(A^*, \preceq)$ is a downward-finite wqo [14].

Well-structured Transition Systems. A well-structured transition system (WSTS for short) is a tuple $(\Sigma, I, \rightarrow, \preceq)$ consisting of a set Σ of states, a finite set $I \subseteq \Sigma$ of initial states, a transition relation $\rightarrow \subseteq \Sigma \times \Sigma$, and a well-quasi-ordering $\preceq \subseteq \Sigma \times \Sigma$ with the monotonicity property: for all $s_1, s_2, t_1 \in \Sigma$ such that $s_1 \rightarrow s_2$ and $s_1 \preceq t_1$, there exists t_2 such that $t_1 \rightarrow t_2$ and $s_2 \preceq t_2$. A WSTS is downward-finite if $(\Sigma, \preceq)$ is downward-finite.

Let $x, y \in \Sigma$. If $x \rightarrow y$, we call x a *predecessor* of y, and y a *successor* of x. We write $\mathrm{pre}(x) := \{y \mid y \rightarrow x\}$ for the *set of predecessors* of x, and $\mathrm{post}(x) := \{y \mid x \rightarrow y\}$ for the *set of successors* of x. For $X \subseteq \Sigma$, $\mathrm{pre}(X)$ and $\mathrm{post}(X)$ are defined as natural extensions, i.e., $\mathrm{pre}(X) = \bigcup_{x \in X} \mathrm{pre}(x)$ and $\mathrm{post}(X) = \bigcup_{x \in X} \mathrm{post}(x)$.

We write $x \rightarrow^k y$ if there are states $x_0, \ldots, x_k \in \Sigma$ such that $x_0 = x$, $x_k = y$ and $x_i \rightarrow x_{i+1}$ for $0 \leq i < k$. Furthermore, $x \rightarrow^* y$ iff there exists a $k \geqslant 0$ such that $x \rightarrow^k y$, i.e., $\rightarrow^*$ is the reflexive and transitive closure of $\rightarrow$. We say that there is a *path of length k from x to y* if $x \rightarrow^k y$, and that there is a *path from x to y* if $x \rightarrow^* y$.

The set of *k-reachable* states Reach_k is the set of states reachable in at most k steps, formally, $\mathsf{Reach}_k := \{y \in \Sigma \mid \exists k' \leqslant k, \exists x \in I, x \rightarrow^{k'} y\}$. The set of *reachable* states $\mathsf{Reach} := \bigcup_{k \geq 0} \mathsf{Reach}_k = \{y \mid \exists x \in I, x \rightarrow^* y\}$. Using downward closure, we can define the *k-th cover* Cover_k and the *cover* Cover of the WSTS as $\mathsf{Cover}_k := \mathsf{Reach}_k{\downarrow}$ and $\mathsf{Cover} := \mathsf{Reach}{\downarrow}$. The *coverability problem for WSTS* asks, given a WSTS $(\Sigma, I, \rightarrow, \preceq)$ and a downward-closed set $P^{\downarrow}$, if every reachable state is contained in $P^{\downarrow}$, i.e., if $\mathsf{Reach} \subseteq P^{\downarrow}$. It is easy to see that this question is equivalent to checking if $\mathsf{Cover} \subseteq P^{\downarrow}$.

In the following, we make some standard effectiveness assumptions on WSTS [1,12]. We assume that $\preceq$ is decidable, and that for any state $x \in \Sigma$, there is a computable procedure that returns a finite basis for $\mathrm{pre}(x{\uparrow})$. These assumptions are met by most classes of WSTS considered in verification [12].

Under the preceding effectiveness assumptions, one can show that the coverability problem is decidable for WSTS by a backward-search algorithm [1]. The main construction is the following sequence of upward-closed sets:

$$\mathsf{U}_0^{\uparrow} := \Sigma \setminus P^{\downarrow}\,, \qquad \mathsf{U}_{i+1}^{\uparrow} := \mathsf{U}_i^{\uparrow} \cup \mathrm{pre}(\mathsf{U}_i^{\uparrow})\,. \qquad \text{(BackwardReach)}$$

The sequence of sets $\mathsf{U}_i^{\uparrow}$ forms an increasing chain of upward-closed sets, therefore it eventually stabilizes: there is some L such that $\mathsf{U}_L^{\uparrow} = \mathsf{U}_{L+i}^{\uparrow}$ for all $i \geq 0$. Then, $\mathsf{Cover} \subseteq P^{\downarrow}$ iff $I \cap \mathsf{U}_L^{\uparrow} = \emptyset$. Moreover, if $I \cap \mathsf{U}_L^{\uparrow} = \emptyset$, then $\Sigma \setminus \mathsf{U}_L^{\uparrow}$ contains I, is contained in $P^{\downarrow}$ and satisfies $\mathrm{post}(\Sigma \setminus \mathsf{U}_L^{\uparrow}) \subseteq \Sigma \setminus \mathsf{U}_L^{\uparrow}$.

We generalize from $\Sigma \setminus \mathsf{U}_L^{\uparrow}$ to the notion of an (inductive) *covering set*. A downward-closed set $C^{\downarrow}$ is called a *covering set* for $P^{\downarrow}$ iff (a) $I \subseteq C^{\downarrow}$, (b) $C^{\downarrow} \subseteq P^{\downarrow}$, and (c) $\mathrm{post}(C^{\downarrow}) \subseteq C^{\downarrow}$. By induction, we have $\mathsf{Cover} \subseteq C^{\downarrow} \subseteq P^{\downarrow}$ for any covering set $C^{\downarrow}$. Therefore, to solve the coverability problem, it is sufficient to exhibit any covering set.

3 IC3 for Coverability

We now describe an algorithm for the coverability problem that takes as input a WSTS $(\Sigma, I, \rightarrow, \preceq)$ and a downward-closed set $P^{\downarrow}$, and constructs either a path from some state in I to a state not in $P^{\downarrow}$ (if $\mathsf{Cover} \not\subseteq P^{\downarrow}$), or a covering set for $P^{\downarrow}$. In the algorithm we consider sets that are not necessarily inductive by themselves, but they are *inductive relative to* some other sets. Relative inductivity is a weakening of the regular notion of inductivity. Formally, for a set $R^{\downarrow}$ such that $I \subseteq R^{\downarrow}$, a downward-closed set $S^{\downarrow}$ is inductive relative to $R^{\downarrow}$ if $I \subseteq S^{\downarrow}$ and $\mathrm{post}(R^{\downarrow} \cap S^{\downarrow}) \subseteq S^{\downarrow}$. An upward-closed set $U^{\uparrow}$ is inductive relative to $R^{\downarrow}$ if its downward-closed complement $\Sigma \setminus U^{\uparrow}$ is inductive relative to $R^{\downarrow}$, i.e. if $I \cap U^{\uparrow} = \emptyset$ and $\mathrm{post}(R^{\downarrow} \setminus U^{\uparrow}) \subseteq \Sigma \setminus U^{\uparrow}$.

The condition $\mathrm{post}(R^{\downarrow} \cap S^{\downarrow}) \subseteq S^{\downarrow}$ is equivalent to $\mathrm{pre}(\Sigma \setminus S^{\downarrow}) \cap R^{\downarrow} \cap S^{\downarrow} = \emptyset$. Stated in terms of an upward-closed set $U^{\uparrow}$, the equivalent condition is $\mathrm{pre}(U^{\uparrow}) \cap R^{\downarrow} \setminus U^{\uparrow} = \emptyset$. Since all these conditions are equivalent, we will use them interchangeably.

3.1 Algorithm

The core idea of the algorithm is to build a vector $\mathbf{R} = (R_0^{\downarrow}, \ldots, R_N^{\downarrow})$, consisting of sets $R_i^{\downarrow}$ which over-approximate Cover_i. The algorithm ensures that $R_{i+1}^{\downarrow}$ is inductive relative to $R_i^{\downarrow}$ in each step, and that $R_i^{\downarrow} \subseteq P^{\downarrow}$ for $i < N$. This allows us to prove that if the vector stabilizes, we have found an inductive covering set.

The algorithm alternates between trying to find a counter-example and refining the over-approximations. A backward search looks for counter-examples. Whenever a candidate counter-example is revealed to be spurious, the vector $\mathbf{R}$ is refined to exclude this counter-example. In particular, counter-example traces are constructed such that if $a_0 \cdots a_N$ is a counter-example, then $a_i \in R_i^{\downarrow}$. If it can be shown that $a_i \in R_i^{\downarrow}$, but the counter-example cannot be extended backwards beyond $R_i^{\downarrow}$, the set $R_i^{\downarrow}$ is refined. In particular, some state b is found such that removing $b\uparrow$ from $R_0^{\downarrow}, \ldots, R_i^{\downarrow}$ gives a new over-approximation vector that still satisfies all invariants, and $a \in b\uparrow$.

When no progress can be made in refining the current vector or in finding counter-examples, N is increased, and a second strengthening step is carried

$$[\mathsf{Initialize}]\ \frac{}{\mathsf{init} \mapsto I{\downarrow} \mid \varnothing} \qquad [\mathsf{Valid}]\ \frac{R_i^{\downarrow} = R_{i+1}^{\downarrow} \text{ for some } i < N}{\mathbf{R} \mid Q \mapsto \mathsf{valid}}$$

$$[\mathsf{CandidateNondet}]\ \frac{a \in R_N^{\downarrow} \setminus P^{\downarrow}}{\mathbf{R} \mid \varnothing \mapsto \mathbf{R} \mid \langle a, N\rangle} \qquad [\mathsf{Model}]\ \frac{\min Q = \langle a, 0\rangle}{\mathbf{R} \mid Q \mapsto \mathsf{invalid}}$$

$$[\mathsf{DecideNondet}]\ \frac{\min Q = \langle a, i\rangle \quad i > 0 \quad b \in \mathrm{pre}(a{\uparrow}) \cap R_{i-1}^{\downarrow} \setminus a{\uparrow}}{\mathbf{R} \mid Q \mapsto \mathbf{R} \mid Q.\textsc{Push}(\langle b, i-1\rangle)}$$

$$[\mathsf{Conflict}]\ \frac{\min Q = \langle a, i\rangle \quad i > 0 \quad \mathrm{pre}(a{\uparrow}) \cap R_{i-1}^{\downarrow} \setminus a{\uparrow} = \emptyset \quad b \in \mathrm{Gen}_{i-1}(a)}{\mathbf{R} \mid Q \mapsto \mathbf{R}[R_k^{\downarrow} \leftarrow R_k^{\downarrow} \setminus b{\uparrow}]_{k=1}^{i} \mid Q.\textsc{PopMin}}$$

$$[\mathsf{Induction}]\ \frac{R_i^{\downarrow} = \Sigma \setminus \{r_{i,1}, \ldots, r_{i,m}\}{\uparrow} \quad b \in \mathrm{Gen}_i(r_{i,j}) \text{ for some } 1 \le j \le m}{\mathbf{R} \mid \varnothing \mapsto \mathbf{R}[R_k^{\downarrow} \leftarrow R_k^{\downarrow} \setminus b{\uparrow}]_{k=1}^{i+1} \mid \varnothing}$$

$$[\mathsf{Unfold}]\ \frac{R_N^{\downarrow} \subseteq P^{\downarrow}}{\mathbf{R} \mid \varnothing \mapsto \mathbf{R} \cdot \Sigma \mid \varnothing}$$

Fig. 1. The rule system for a IC3-style algorithm for WSTS – generic version. The map Gen_i is defined in equation (1).

out, in which states in $R_i^{\downarrow}$ that can be proven to be unreachable from $R_{i-1}^{\downarrow}$ are removed. This strengthening step is known as induction.

Figure 1 shows the algorithm as a set of non-deterministic state transition rules, similar to [15]. A state of the computation is either the initial state init, the special terminating states valid and $\mathsf{invalid}$, or a pair $\mathbf{R} \mid Q$ defined as follows.

The first component of the pair is a vector $\mathbf{R}$ of downward-closed sets, indexed starting from 0. The elements of $\mathbf{R}$ are denoted $R_i^{\downarrow}$. In particular, we denote by $R_0^{\downarrow}$ the downward closure of I, i.e., $R_0^{\downarrow} = I{\downarrow}$. These sets contain successive approximations to a potential covering set. The function length gives the length of the vector, disregarding $R_0^{\downarrow}$, i.e., $\mathrm{length}(R_0^{\downarrow}, \ldots, R_N^{\downarrow}) = N$. If it is clear from the context which vector is meant, we often abbreviate $\mathrm{length}(\mathbf{R})$ simply with N. We write $\mathbf{R} \cdot X$ for the concatenation of the vector $\mathbf{R}$ with the downward-closed set X: $(R_0^{\downarrow}, \ldots, R_N^{\downarrow}) \cdot X = (R_0^{\downarrow}, \ldots, R_N^{\downarrow}, X)$.

The second component of the pair is a priority queue Q, containing elements of the form $\langle a, i\rangle$, where $a \in \Sigma$ is a state and $i \in \mathbb{N}$ is a natural number. The priority of the element is given by i, and is called the level of the element. The statement $\langle a, i\rangle \in Q$ means that the priority queue contains an element $\langle a, i\rangle$, while with $\min Q$ we denote the minimal element of the priority queue. Furthermore, $Q.\textsc{PopMin}$ yields Q after removal of its minimal element, and $Q.\textsc{Push}(x)$ yields Q after adding element x.

The elements of Q are states that lead outside of $P^{\downarrow}$. Let $\langle a, i\rangle$ be an element of Q. Either a is a state that is in R_i and outside of $P^{\downarrow}$, or there is a state b leading outside of $P^{\downarrow}$ such that $a \in \mathrm{pre}(b{\uparrow})$. Our goal is to try to discard those states and show that they are not reachable from the initial states, as R_i denotes

an over-approximation of the states reachable in i or less steps. If an element of Q is reachable from the initial states, then $\mathsf{Cover} \not\subseteq P^{\downarrow}$.

The state valid signifies that the search has terminated with $\mathsf{Cover} \subseteq P^{\downarrow}$, while invalid signifies that the algorithm has terminated with $\mathsf{Cover} \not\subseteq P^{\downarrow}$. In the description of the algorithm, we will omit the actual construction of certificates and instead just state that the algorithm terminates with invalid or valid; the calculation of certificates is straightforward.

The transition rules of the algorithm are of the form

$$[\text{Name}]\frac{C_1 \ \cdots \ C_k}{\sigma \mapsto \sigma'} \qquad \text{(Rule)}$$

and can be read thus: whenever the algorithm is in state σ and conditions C_1–C_k are fulfilled, the algorithm can apply rule [Name] and transition to state σ'. We write $\sigma \mapsto \sigma'$ if there is some rule which the algorithm applies to go from σ to σ'. We write $\mapsto^*$ for the reflexive and transitive closure of $\mapsto$.

Let Inv be a predicate on states. We say that a rule *preserves the invariant* Inv if whenever σ satisfies Inv and conditions C_1 to C_k are met, it also holds that σ' satisfies Inv.

Two of the rules use the map $\mathrm{Gen}_i : \Sigma \to 2^{\Sigma}$. It yields those states that are valid generalizations of a relative to some set $R_i^{\downarrow}$. A state b is a generalization of the state a relative to the set $R_i^{\downarrow}$, if $b \preceq a$ and $b{\uparrow}$ is inductive relative to $R_i^{\downarrow}$. Formally,

$$\mathrm{Gen}_i(a) := \{b \mid b \preceq a \wedge b{\uparrow} \cap I = \emptyset \wedge \mathrm{pre}(b{\uparrow}) \cap R_i^{\downarrow} \setminus b{\uparrow} = \emptyset\}\,. \qquad (1)$$

Finally, the notation $\mathbf{R}[R_k^{\downarrow} \leftarrow {R'}_k^{\downarrow}]_{k=1}^{i}$ means that $\mathbf{R}$ is transformed by replacing $R_k^{\downarrow}$ by ${R'}_k^{\downarrow}$ for each $k = 1, \ldots, i$, i.e.,

$$\mathbf{R}[R_k^{\downarrow} \leftarrow {R'}_k^{\downarrow}]_{k=1}^{i} = (R_0^{\downarrow}, {R'}_1^{\downarrow}, \ldots, {R'}_i^{\downarrow}, R_{i+1}^{\downarrow}, \ldots, R_N^{\downarrow}).$$

We provide an overview of each rule of the calculus.

[Initialize] The algorithm starts by defining the first downward-closed set $R_0^{\downarrow}$ to be the downward closure of the initial state.

[CandidateNondet] If there is a state a such that $a \in R_N^{\downarrow}$, but at the same time it is not an element of $P^{\downarrow}$, we add $\langle a, N\rangle$ to the priority queue Q.

[DecideNondet] To check if the elements of Q are spurious counter-examples, we start by processing an element a with the lowest level i. If there is an element b in $R_{i-1}^{\downarrow}$ such that $b \in \mathrm{pre}(a{\uparrow})$, then we add $\langle b, i-1\rangle$ to the priority queue.

[Model] If Q contains a state a from the level 0, then we have found a counter-example trace and the algorithm terminates in the state invalid.

[Conflict] If none of predecessors of a state a from the level i is contained in $R_{i-1}^{\downarrow} \setminus a{\uparrow}$, then a belongs to a spurious counter-example trace. Therefore, we update the downward-closed sets $R_1^{\downarrow}, \ldots, R_i^{\downarrow}$ as follows: since the states in $a{\uparrow}$ are not reachable in i steps, they can be safely removed from all the sets $R_1^{\downarrow}, \ldots, R_i^{\downarrow}$. Moreover, instead of $a{\uparrow}$ we can remove even a bigger set

$b\uparrow$, for any state b which is a generalization of the state a relative to $R_{i-1}^{\downarrow}$, as defined in (1). After the update, we remove $\langle a, i\rangle$ from the priority queue. In practice, if $i < N$, we also add $\langle a, i+1\rangle$ to the priority queue. This allows the construction of paths that are longer than N in a given search, which speeds up search significantly (see [3]). It also justifies the use of a priority queue for Q, instead of a stack. We omit this modification in our rules to simplify proofs.

[Induction] If for some state $r_{i,j}$ that was previously removed from $R_i^{\downarrow}$, a set $r_{i,j}\uparrow$ becomes inductive relative to $R_i^{\downarrow}$ (i.e. $\mathrm{post}(R_i^{\downarrow} \setminus r_{i,j}\uparrow) \subseteq \Sigma \setminus r_{i,j}\uparrow$), none of the states in $r_{i,j}\uparrow$ can be reached in at most $i+1$ steps. Thus, we can safely remove $r_{i,j}\uparrow$ from $R_{i+1}^{\downarrow}$ as well. Similarly as in [Conflict], we can even remove $b\uparrow$ for any generalization $b \in \mathrm{Gen}_i(r_{i,j})$.

[Valid] If there is a downward-closed set $R_i^{\downarrow}$ such that $R_i^{\downarrow} = R_{i+1}^{\downarrow}$, the algorithm terminates in the state valid.

[Unfold] If the priority queue is empty and all elements of $R_N^{\downarrow}$ are in $P^{\downarrow}$, we start with a construction of the next set $R_{N+1}^{\downarrow}$. Initially, $R_{N+1}^{\downarrow}$ contains all the states, $R_{N+1}^{\downarrow} = \Sigma$, and we append $R_{N+1}^{\downarrow}$ to the vector $\mathbf{R}$.

In an implementation, these rules are usually applied in a specific way. The algorithm generally proceeds in rounds consisting of two phases: backward search and inductive strengthening.

In the backward search phase, the algorithm tries to construct a path $a_0 \cdots a_N$ from I to $\Sigma \setminus P^{\downarrow}$, with $a_i \in R_i^{\downarrow}$. The path is built backwards from $R_N^{\downarrow}$, by first finding a possible end-point using [CandidateNondet]. Next, [DecideNondet] is applied to prolong the path, until $R_0^{\downarrow}$ is reached (as witnessed by [Model]). If the path cannot be prolonged at a_i, [Conflict] is applied to refine the sets $R_1^{\downarrow}, \ldots, R_i^{\downarrow}$, removing known-unreachable states (including a_i), and the search backtracks one step. If the backward search phase ends unsuccessfully, i.e. no more paths can be constructed, [Unfold] is applied.

In the inductive strengthening phase, [Induction] is repeatedly applied until its pre-conditions fail to hold. After that, it is checked whether [Valid] applies. If not, the algorithm proceeds to the next round.

3.2 Soundness

We first show that the algorithm is sound: if it terminates, it produces the right answer. If it terminates in the state invalid, there is a path from an initial state to a state outside of $P^{\downarrow}$, and if it terminates in the state valid, then $\mathsf{Cover} \subseteq P^{\downarrow}$.

We prove soundness by showing that on each state $\mathbf{R} \mid Q$ the following invariants are preserved by the transition rules:

$$I \subseteq R_i^{\downarrow} \qquad \text{for all } 0 \le i \le N \qquad \text{(I1)}$$

$$\mathrm{post}(R_i^{\downarrow}) \subseteq R_{i+1}^{\downarrow} \qquad \text{for all } 0 \le i < N \qquad \text{(I2)}$$

$$R_i^{\downarrow} \subseteq R_{i+1}^{\downarrow} \qquad \text{for all } 0 \le i < N \qquad \text{(I3)}$$

$$R_i^{\downarrow} \subseteq P^{\downarrow} \qquad \text{for all } 0 \le i < N \qquad \text{(I4)}$$

These properties imply $R_i^{\downarrow} \supseteq \mathsf{Cover}_i$, that is, the region R_i provides an over-approximation of the i-cover.

The first step of the algorithm (rule [Initialize]) results with the state $I{\downarrow} \mid \varnothing$, which satisfies (I2)–(I4) trivially, and $I \subseteq I{\downarrow}$ establishes (I1). The following lemma states that the invariants are preserved by rules that do not result in valid or invalid.

Lemma 1. *The rules* [Unfold], [Induction], [Conflict], [CandidateNondet], *and* [DecideNondet] *preserve* (I1) – (I4),

By induction on the length of the trace, it can be shown that if $\mathsf{init} \mapsto^* \mathbf{R} \mid Q$, then $\mathbf{R} \mid Q$ satisfies (I1) – (I4). When $\mathsf{init} \mapsto^* \mathsf{valid}$, there is a state $\mathbf{R} \mid Q$ such that $\mathsf{init} \mapsto^* \mathbf{R} \mid Q \mapsto \mathsf{valid}$, and the last applied rule is [Valid]. To be able to apply [Valid], there must be an i such that $R_i^{\downarrow} = R_{i+1}^{\downarrow}$.

We claim that $R_i^{\downarrow}$ is a covering set. This claim follows from (1) $R_i^{\downarrow} \subseteq P^{\downarrow}$ by invariant (I4), (2) $I \subseteq R_i^{\downarrow}$ by invariant (I1), and (3) $\mathrm{post}(R_i^{\downarrow}) \subseteq R_{i+1}^{\downarrow} = R_i^{\downarrow}$ by invariant (I2). This proves the correctness of the algorithm in case $\mathsf{Cover} \subseteq P^{\downarrow}$:

Theorem 1 (Soundness of uncoverability). *Given a WSTS* $(\Sigma, I, \rightarrow, \preceq)$ *and a downward-closed set* $P^{\downarrow}$*, if* $\mathsf{init} \mapsto^* \mathsf{valid}$*, then* $\mathsf{Cover} \subseteq P^{\downarrow}$.

We next consider the case when $\mathsf{Cover} \not\subseteq P^{\downarrow}$. The following lemma describes an invariant of the priority queue.

Lemma 2. *Let* $\mathsf{init} \mapsto^* \mathbf{R} \mid Q$. *For every* $\langle a, i\rangle \in Q$, *there is a path from* a *to some* $b \in \Sigma \setminus P^{\downarrow}$.

Theorem 2 (Soundness of coverability). *Given a WSTS* $(\Sigma, I, \rightarrow, \preceq)$ *and a downward-closed set* $P^{\downarrow}$*, if* $\mathsf{init} \mapsto^* \mathsf{invalid}$*, then* $\mathsf{Cover} \not\subseteq P^{\downarrow}$.

Proof. The assumption $\mathsf{init} \mapsto^* \mathsf{invalid}$ implies that there is some state $\mathbf{R} \mid Q$ such that $\mathsf{init} \mapsto^* \mathbf{R} \mid Q \mapsto \mathsf{invalid}$, and the last applied rule was [Model]. Therefore, there is an a such that $\langle a, 0\rangle \in Q$. By Lemma 2, there is a path from a to some $b \in \Sigma \setminus P^{\downarrow}$. Since $a \in I{\downarrow}$, we have $b \in \mathsf{Cover}$. □

3.3 Termination

While the above non-deterministic rules guarantee soundness for any WSTS, termination requires some additional choices. We modify the [DecideNondet] and [CandidateNondet] rules into more restricted rules [Decide] and [Candidate], shown in Figure 2. All other rules are unchanged.

The restricted rules still preserve the invariants (I1) – (I4). Thus, the modified algorithm is still sound. To show termination, we first note that the system can make progress until it reaches either valid or invalid.

Proposition 1 (Progress). *If* $\mathsf{init} \mapsto^* \mathbf{R} \mid Q$, *then one of the rules* [Candidate], [Decide], [Conflict], [Induction], [Unfold], [Model] *and* [Valid] *is applicable.*

$$[\mathsf{Candidate}]\frac{a \in R_N^{\downarrow} \cap \min(\Sigma \setminus P^{\downarrow})}{\mathbf{R} \mid \varnothing \mapsto \mathbf{R} \mid \langle a, N\rangle}$$

$$[\mathsf{Decide}]\frac{\min Q = \langle a, i\rangle \quad i > 0 \quad b \in \min(\mathrm{pre}(a\uparrow)) \cap R_{i-1}^{\downarrow} \setminus a\uparrow}{\mathbf{R} \mid Q \mapsto \mathbf{R} \mid Q.\textsc{Push}(\langle b, i-1\rangle)}$$

Fig. 2. Rules replacing [CandidateNondet] and [DecideNondet] in Fig. 1.

Next, we define an ordering on states $\mathbf{R} \mid Q$.

Definition 1. *Let* $\mathbf{A}^{\downarrow} = (A_1^{\downarrow}, \ldots, A_N^{\downarrow})$ *and* $\mathbf{B}^{\downarrow} = (B_1^{\downarrow}, \ldots, B_N^{\downarrow})$ *be two finite sequences of downward-closed sets of the equal length* N*. Define* $\mathbf{A}^{\downarrow} \sqsubseteq \mathbf{B}^{\downarrow}$ *iff* $A_i^{\downarrow} \subseteq B_i^{\downarrow}$ *for all* $i = 1, \ldots, N$*. Let* Q *be a priority queue whose elements are tuples* $\langle a, i\rangle \in \Sigma \times \{0, \ldots, N\}$*. Define* $\ell_N(Q) := \min(\{i \mid \langle a, i\rangle \in Q\} \cup \{N+1\})$*, to be the smallest priority in* Q*, or* $N+1$ *if* Q *is empty.*

For two states $\mathbf{R} \mid Q$ *and* $\mathbf{R}' \mid Q'$ *such that* $\mathrm{length}(\mathbf{R}) = \mathrm{length}(\mathbf{R}') = N$*, we define the ordering* $\leq_s$ *as:*

$$\mathbf{R} \mid Q \leq_s \mathbf{R}' \mid Q' : \Longleftrightarrow\ \mathbf{R} \sqsubseteq \mathbf{R}' \wedge (\mathbf{R} = \mathbf{R}' \rightarrow \ell_N(Q) \leq \ell_N(Q'))\,.$$

We write $\mathbf{R} \mid Q <_s \mathbf{R}' \mid Q'$ *if* $\mathbf{R} \mid Q \leq_s \mathbf{R}' \mid Q'$ *and* $\mathbf{R}' \mid Q' \not\leq_s \mathbf{R} \mid Q$*.*

Lemma 3. *Given a natural number* N*, the relation* $\leq_s$ *is a well-quasi-ordering on the set* $\mathcal{D}^N \times \mathcal{Q}_N$*, where* $\mathcal{D}$ *is a set of downward-closed subsets of* Σ*, and* $\mathcal{Q}_N$ *denotes the set of priority queues over* $\Sigma \times \{0, \ldots, N\}$*.*

Using the well-quasi-ordering $\leq_s$, we can prove a lemma which characterizes infinite runs of the algorithm. The proof follows from the observation that if $\mathbf{R} \mid Q \mapsto \mathbf{R}' \mid Q'$ as a result of applying the [Candidate], [Decide], [Conflict], or [Induction] rules, then $\mathbf{R}' \mid Q' <_s \mathbf{R} \mid Q$.

Lemma 4 (Infinite sequence condition). *Any infinite sequence* $\mathit{init} \mapsto \sigma_1 \mapsto \sigma_2 \mapsto \cdots$*, must contain infinitely many* i *such that* $\sigma_i \mapsto \sigma_{i+1}$ *by applying the rule* [Unfold]*.*

Note that applying the rule [Unfold] increases the length of the sequence $\mathbf{R}$. Therefore, in any infinite run of the algorithm, $\mathrm{length}(\mathbf{R})$ increases unboundedly. On the other hand, only a finite number of different sets $R_i^{\downarrow}$ can be generated for a downward-finite WSTS. To show that, we define a sequence of sets D_i, which provide a finite representation of states backward reachable from $\Sigma \setminus P^{\downarrow}$.

Recall the sequence $\mathsf{U}_i^{\uparrow}$ of backward reachable states from (BackwardReach). The set D_i captures all new elements that are introduced in $\mathsf{U}_i^{\uparrow}$ and that were not present in the previous iterations. Formally, we define sets D_i as follows:

$$D_0 := \min(\Sigma \setminus P^{\downarrow})\,, \qquad D_{i+1} := \bigcup_{a \in D_i} \min(\mathrm{pre}(a\uparrow)) \setminus \mathsf{U}_i^{\uparrow}\,. \tag{2}$$

By induction and the finiteness of the set of minimal elements, it follows that D_i is finite for all $i \geqslant 0$. Furthermore, there is an L such that $\mathsf{U}^{\uparrow}_{L} = \mathsf{U}^{\uparrow}_{L+j}$, and therefore $D_{L+j} = \emptyset$, for all $j > 0$. As a consequence, the set $\bigcup_{i \geqslant 0} D_i$ is finite.

Lemma 5. *Let* $\mathit{init} \mapsto^* \mathbf{R} \mid Q$. *For every* $\langle a, i \rangle \in Q$, *it holds that* $a \in D_{N-i}$.

Lemma 6. *Given a downward-finite WSTS* $(\Sigma, I, \to, \preceq)$ *and a downward-closed set* $P^{\downarrow}$, *let* $D := \bigcup_{i \geqslant 0} D_i$. *Then there is* N_0 *such that for any sequence* $\mathbf{R}$ *of* $\mathrm{length}(\mathbf{R}) = N > N_0$, *generated by applying the restricted rules, there is* $i < N$ *such that* $R_i^{\downarrow} = R_{i+1}^{\downarrow}$.

Proof. From Lemma 5 and the definition (1) of Gen_i, it follows that the restricted rules generate sequences $\mathbf{R}$ such that $R_i^{\downarrow} = \Sigma \setminus B_i{\uparrow}$, $B_i \supseteq B_{i+1}$ and $B_i \subseteq D{\downarrow}$, for $i > 0$. Since D is finite, $D{\downarrow}$ is also finite by downward-finiteness. Hence, there is only a finite number of possible sets of the form $\Sigma \setminus B{\uparrow}$, for $B \subseteq D{\downarrow}$. Therefore, sequences $\mathbf{R}$ of sufficient length contain equal adjacent sets. □

Combining Lemmas 4 and 6, we see that in any infinite run of the algorithm, the rule [Valid] becomes applicable from some point onward. If we impose a fair usage of that rule (i.e. the rule is eventually used after it becomes applicable), we get termination.

Theorem 3 (Termination). *Given a downward-finite WSTS* $(\Sigma, I, \to, \preceq)$ *and a downward-closed set* $P^{\downarrow}$, *if the rule* [Valid] *is used fairly, the algorithm reaches either* valid *or* invalid.

Note that Theorem 3 is the only result that requires downward-finiteness of the WSTS. We show that the downward-finiteness condition is necessary for the termination of the abstract IC3 algorithm, using arbitrary generalization schemes and the full leeway in applying the rules. Consider a WSTS $(\mathbb{N} \cup \{\omega\}, \{0\}, \to, \leqslant)$, where $x \to x + 1$ for each $x \in \mathbb{N}$ and $\omega \to \omega$, and $\leqslant$ is the natural order on $\mathbb{N}$ extended with $x \leqslant \omega$ for all $x \in \mathbb{N}$. Consider the downward-closed set $\mathbb{N}$. The backward analysis terminates in one step, since $\mathrm{pre}(\omega) = \{\omega\}$. However, the IC3 algorithm need not terminate. After unfolding, we find a conflict since $\mathrm{pre}(\omega) = \{\omega\}$, which is not initial. Generalizing, we get $R_1^{\downarrow} = \{0, 1\}$. At this point, we unfold again. We find another conflict, and generalize to $R_2^{\downarrow} = \{0, 1, 2\}$. We continue this way to generate an infinite sequence of steps without terminating.

4 Coverability for Petri Nets

We now describe an implementation of our algorithm for the coverability problem for Petri nets, a widely used model for concurrent systems.

4.1 Petri Nets

A Petri net (PN, for short) is a tuple (S, T, W), where S is a finite set of *places*, T is a finite set of *transitions* disjoint from S, and $W : (S \times T) \cup (T \times S) \to \mathbb{N}$ is the arc multiplicity function.

The semantics of a PN is given using *markings*. A marking is a function from S to $\mathbb{N}$. For a marking m and place $s \in S$, we say s has $m(s)$ tokens. A transition $t \in T$ is *enabled* at marking m, written $m|t\rangle$, if $m(s) \geqslant W(s,t)$ for all $s \in S$. A transition t that is enabled at m can fire, yielding a new marking m' such that $m'(s) = m(s) - W(s,t) + W(t,s)$. We write $m|t\rangle m'$ to denote the transition from m to m' on firing t.

A PN (S, T, W) and an initial marking m_0 give rise to a WSTS $(\Sigma, \{m_0\}, \rightarrow, \preceq)$, where Σ is the set of markings, and m_0 is a single initial state. The transition relation is defined as follows: there is an edge $m \rightarrow m'$ if and only if there is some transition $t \in T$ such that $m|t\rangle m'$. The well-quasi-ordering satisfies the following property: $m \preceq m'$ if and only if for each $s \in S$ we have $m(s) \leqslant m'(s)$. The compatibility condition holds: if $m_1|t\rangle m_2$ and $m_1 \preceq m_1'$, then there is a marking m_2' such that $m_1'|t\rangle m_2'$ and $m_2 \preceq m_2'$. Moreover, since markings can be only non-negative integers, the wqo is downward-finite. The coverability problem for PNs is defined as the coverability problem on this WSTS.

We represent Petri nets as follows. Let $S = \{s_1, \ldots, s_n\}$ be the set of places. A marking m is represented as the tuple of natural numbers $(m(s_1), \ldots, m(s_n))$. A transition t is represented as a pair $(\mathbf{g}, \mathbf{d}) \in \mathbb{N}^n \times \mathbb{Z}^n$, where $\mathbf{g}$ represents the enabling condition, and $\mathbf{d}$ represents the difference between the number of tokens in a place if the transition fires, and the current number of tokens. Formally, $\mathbf{g} = (W(s_1,t), \ldots, W(s_n,t))$ and $\mathbf{d} = (W(t,s_1) - W(s_1,t), \ldots, W(t,s_n) - W(s_n,t))$.

We represent upward-closed sets with their minimal bases, which are finite sets of n-tuples of natural numbers. A downward-closed set is represented as its complement (which is an upward-closed set). The sets $R_i^{\downarrow}$, which are constructed during the algorithm run, are therefore represented as their complements. Such a representation comes naturally as the algorithm executes. Originally each set $R_i^{\downarrow}$ is initialized to contain all the states. The algorithm removes sets of states of the form $\mathbf{b}\uparrow$ from $R_i^{\downarrow}$, for some $\mathbf{b} \in \mathbb{N}^n$. If a set $\mathbf{b}\uparrow$ was removed from $R_i^{\downarrow}$, we say that states in $\mathbf{b}\uparrow$ are *blocked* by $\mathbf{b}$ at level i. At the end every $R_i^{\downarrow}$ becomes to a set of the form $\Sigma \setminus \{\mathbf{b}_1, \ldots, \mathbf{b}_l\}\uparrow$ and we conceptually represent $R_i^{\downarrow}$ with $\{\mathbf{b}_1, \ldots, \mathbf{b}_l\}$.

The implementation uses a succinct representation of $\mathbf{R}$, so called *delta-encoding* [8]. Let $R_i^{\downarrow} = \Sigma \setminus B_i\uparrow$ and $R_{i+1}^{\downarrow} = \Sigma \setminus B_{i+1}\uparrow$ for some finite sets B_i and B_{i+1}. Applying the invariant (I3) yields $B_{i+1} \subseteq B_i$. Therefore we only need to maintain a vector $\mathbf{F} = (F_0, \ldots, F_N, F_\infty)$ such that $\mathbf{b} \in F_i$ if i is the highest level where $\mathbf{b}$ was blocked. This is sufficient because $\mathbf{b}$ is also blocked on all lower levels. As an illustration, for $(R_0^{\downarrow}, R_1^{\downarrow}, R_2^{\downarrow}) = (\{\mathbf{i}_1, \mathbf{i}_2\}, \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3, \mathbf{b}_4\}, \{\mathbf{b}_2, \mathbf{b}_3\})$, the matching vector $\mathbf{F}$ might be $(F_0, F_1, F_2, F_\infty) = (\{\mathbf{i}_1, \mathbf{i}_2\}, \{\mathbf{b}_1, \mathbf{b}_4\}, \{\mathbf{b}_2, \mathbf{b}_3\}, \emptyset)$. The set F_∞ represents states that can never be reached.

4.2 Implementation Details and Optimizations

Our implementation follows the rules given in Figures 1 and 2. In addition, we use optimizations from [8]. The main difference between our implementation and [8] is in the interpretation of sets being blocked: in [8] those are cubes

identified with partial assignments to boolean variables, whereas in our case those are upward-closed sets generated by a single state. Also, a straightforward adaptation of the implementation [8] would replace a SAT solver with a solver for integer difference logic, a fragment of linear integer arithmetic which allows the most natural encoding of Petri nets. However, we observed that Petri nets allow an easy and efficient way of computing predecessors and deciding relative inductiveness directly. Thus we were able to eliminate the overhead of calling the SMT solver.

Testing Membership in $R_i^{\downarrow}$. Many of the rules given in Figures 1 and 2 depend on testing whether some state $\mathbf{a}$ is contained in a set $R_k^{\downarrow}$. Using the delta-encoded vector $\mathbf{F}$ this can be done by iterating over F_i for $k \leqslant i \leqslant N+1$ and checking if any of them contains a state $\mathbf{c}$ such that $\mathbf{c} \preceq \mathbf{a}$. If there is such a state, it blocks $\mathbf{a}$, otherwise $\mathbf{a} \in R_k^{\downarrow}$. If $k = 0$, we search for $\mathbf{c}$ only in F_0.

Implementation of the Rules. The delta-encoded representation $\mathbf{F}$ also makes [Valid] easy to implement. Checking if $R_i^{\downarrow} = R_{i+1}^{\downarrow}$ reduces to checking if F_i is empty for some $i < N$. [Unfold] is applied when [Candidate] can no longer yield a bad state contained in $R_N^{\downarrow}$. It increases N and inserts an empty set to position N in the vector $\mathbf{F}$, thus pushing F_∞ from position N to $N+1$. We implemented rules [Initialize], [Candidate] and [Model] in a straightforward manner.

Computing Predecessors. In the rest of the rules we need to find predecessors $\mathrm{pre}(\mathbf{a}\uparrow)$ in $R_i^{\downarrow} \setminus \mathbf{a}\uparrow$, or conclude relative inductiveness if no such predecessors exist. The implementation in [8] achieves this by using a function *solveRelative()* which invokes the SAT solver. But *solveRelative()* also does two important improvements. In case the SAT solver finds a cube of predecessors, it applies *ternary simulation* to expand it further. If the SAT solver concludes relative inductiveness, it extracts information to conclude a generalized clause is inductive relative to some level $k \geqslant i$. We succeeded to achieve analogous effects in case of Petri nets by the following observations. While it is unclear what ternary simulation would correspond to for Petri nets, the following lemma shows how to compute the most general predecessor along a fixed transition directly.

Lemma 7. *Let $\mathbf{a} \in \mathbb{N}^n$ be a state and $t = (\mathbf{g}, \mathbf{d}) \in \mathbb{N}^n \times \mathbb{Z}^n$ be a transition. Then $\mathbf{b} \in \mathrm{pre}(\mathbf{a}\uparrow)$ is a predecessor along t if and only if $\mathbf{b} \succeq \max(\mathbf{a} - \mathbf{d}, \mathbf{g})$.*

Therefore, to find an element of $R_i^{\downarrow} \setminus \mathbf{a}\uparrow$ and $R_i^{\downarrow} \setminus \mathbf{a}\uparrow$, we iterate through all transitions $t = (\mathbf{g}, \mathbf{d})$ and find the one for which $\max(\mathbf{a} - \mathbf{d}, \mathbf{g}) \in R_i^{\downarrow} \setminus \mathbf{a}\uparrow$.

If there are no such transitions, then $\mathbf{a}\uparrow$ is inductive relative to $R_i^{\downarrow}$. In that case, for each transition $t = (\mathbf{g}, \mathbf{d})$ the predecessor $\max(\mathbf{a} - \mathbf{d}, \mathbf{g})$ is either blocked by $\mathbf{a}$ itself, or there is $i_t \geqslant i$ and a state $\mathbf{c}_t \in F_{i_t}$ such that $\mathbf{c}_t \preceq \max(\mathbf{a} - \mathbf{d}, \mathbf{g})$. We define

$$i' := \min\{i_t \mid t \text{ is a transition}\},$$

where $i_t := N + 1$ for $t = (\mathbf{g}, \mathbf{d})$ if $\max(\mathbf{a} - \mathbf{d}, \mathbf{g})$ is blocked by $\mathbf{a}$ itself. Then $i' \geqslant i$ and $\mathbf{a}\uparrow$ is inductive relative to $R_{i'}^{\downarrow}$.

Computing Generalizations. The following lemma shows that we can also significantly generalize $\mathbf{a}$, i.e. there is a simple way to compute a state $\mathbf{a}' \preceq \mathbf{a}$ such

that for all transitions $t = (\mathbf{g}, \mathbf{d})$, $\max(\mathbf{a}' - \mathbf{d}, \mathbf{g})$ remains blocked either by $\mathbf{a}'$ itself, or by $\mathbf{c}_t$.

Lemma 8. *Let $\mathbf{a}, \mathbf{c} \in \mathbb{N}^n$ be states and $t = (\mathbf{g}, \mathbf{d}) \in \mathbb{N}^n \times \mathbb{Z}^n$ be a transition.*

1. *Let $\mathbf{c} \preceq \max(\mathbf{a} - \mathbf{d}, \mathbf{g})$. Define $\mathbf{a}'' \in \mathbb{N}^n$ by $a''_j := c_j + d_j$ if $g_j < c_j$ and $a''_j := 0$ if $g_j \geqslant c_j$, for $j = 1, \ldots, n$. Then $\mathbf{a}'' \preceq \mathbf{a}$. Additionally, for each $\mathbf{a}' \in \mathbb{N}^n$ such that $\mathbf{a}'' \preceq \mathbf{a}' \preceq \mathbf{a}$, we have $\mathbf{c} \preceq \max(\mathbf{a}' - \mathbf{d}, \mathbf{g})$.*
2. *If $\mathbf{a} \preceq \max(\mathbf{a} - \mathbf{d}, \mathbf{g})$, then for each $\mathbf{a}' \in \mathbb{N}^n$ such that $\mathbf{a}' \preceq \mathbf{a}$, it holds that $\mathbf{a}' \preceq \max(\mathbf{a}' - \mathbf{d}, \mathbf{g})$.*

To continue with the case when the predecessor $\max(\mathbf{a} - \mathbf{d}, \mathbf{g})$ is blocked for each transition $t = (\mathbf{g}, \mathbf{d})$, we define $\mathbf{a}''_t$ as in Lemma 8 (1) if the predecessor is blocked by some state $\mathbf{c}_t \in F_{i_t}$ and $\mathbf{a}''_t := (0, \ldots, 0)$ if it is blocked by $\mathbf{a}$ itself. The state $\mathbf{a}''$ is defined to be the pointwise maximum of all states $\mathbf{a}''_t$. By Lemma 8, predecessors of $\mathbf{a}''$ remain blocked by the same states $\mathbf{c}_t$ or by $\mathbf{a}''$ itself.

However, $\mathbf{a}''$ still does not have to be a valid generalization, because it might be in $R_0^{\downarrow}$. If that is the case, we take any state $\mathbf{c} \in F_0$ which blocks $\mathbf{a}$ (such a state exists because $\mathbf{a} \notin R_0^{\downarrow}$). Then $\mathbf{a}' := \max(\mathbf{a}'', \mathbf{c})$ is a valid generalization: $\mathbf{a}' \preceq \mathbf{a}$ and $\mathbf{a}'\uparrow$ is inductive relative to $R_{i'}^{\downarrow}$.

Using this technique, rules [Decide], [Conflict] and [Induction] become easy to implement. Note that some additional handling is needed in rules [Conflict] and [Induction] when blocking a generalized upward-closed set $\mathbf{a}'\uparrow$. If $\mathbf{a}'\uparrow$ is inductive relative to $R_{i'}^{\downarrow}$ for $i' < N$, we update the vector $\mathbf{F}$ by adding $\mathbf{a}'$ to $F_{i'+1}$. However, if $i' = N$ or $i' = N + 1$, we add $\mathbf{a}'$ to $F_{i'}$. Additionaly, for $1 \leqslant k \leqslant i' + 1$ (or $1 \leqslant k \leqslant i'$) we remove all states $\mathbf{c} \in F_k$ such that $\mathbf{a}' \preceq \mathbf{c}$.

5 Experimental Evaluation

We have implemented the IC3 algorithm in a tool called IIC. Our tool is written in C++ and uses the input format of mist2[1]. We evaluated the efficiency of the algorithm on a collection of Petri net examples. The goal of the evaluation was to compare the performance —both time and space usage— of IIC against other implementations of Petri net coverability.

We compare the performance of IIC, using our implementation described above, to the following algorithms: EEC [13] and backward search [1], as implemented by the tool mist2, and the MCOV algorithm [16] for parameterized multithreaded programs as implemented by bfc[2]. All experiments were performed on identical machines, each having Intel Xeon 2.67 GHz CPUs and 48 GB of memory, running Linux 3.2.21 in 64 bit mode. Execution time was limited to 1 hour, and memory to five gigabytes.

Mist2 and MedXXX Benchmarks. We used 29 Petri net examples from the mist2 distribution and 12 examples from checking security properties of message-passing programs communicating through unbounded and unordered channels

[1] See `http://software.imdea.org/~pierreganty/ist.html`
[2] See `http://www.cprover.org/bfc/`

Table 1. Experimental results: comparison of running time and memory consumption for different coverability algorithms on Petri net benchmarks. Memory consumption is in MB, and running time in seconds. In the MCOV column, the superscripts indicate the version of bfc used ([1] means the version Jan 2012 version, [2] the Feb 2013 version), and the analysis mode ([c]: combined, [b]: backward only, [f]: forward only). We list the best result for all the version/parameter combinations that were tried. EEC timed out on all MedXXX examples.

Problem	IIC		Backward		EEC		MCOV	
Instance	Time	Mem	Time	Mem	Time	Mem	Time	Mem
Uncoverable instances								
Bingham ($h = 150$)	**0.1**	**3.5**	970.3	146.3	1.8	19.0	**0.1**	7.6^{2c}
Bingham ($h = 250$)	**0.2**	**6.7**	Timeout		9.6	45.4	**0.2**	19.6^{2c}
Ext. ReadWrite (small consts)	**0.0**	**1.3**	0.1	3.7	Timeout		Timeout/OOM	
Ext. ReadWrite	**0.3**	**1.5**	216.3	34.1	Timeout		0.6	4.1^{2b}
FMS (old)	< **0.1**	**1.3**	1.3	5.5	Timeout		0.1	5.8^{2c}
Mesh2x2	< **0.1**	**1.3**	0.3	3.9	266.9	24.3	< **0.1**	4.2^{1c}
Mesh3x2	< **0.1**	**1.5**	4.1	7.0	Timeout		< **0.1**	2.0^{2b}
Multipoll	1.5	**1.6**	0.5	4.3	21.8	7.1	< **0.1**	1.7^{2b}
MedAA1	**0.5**	**173.3**	8.8	598.8			3.7	210.4^{2b}
MedAA2	Timeout		Timeout				Timeout/OOM	
MedAA5	Timeout		Timeout				Timeout/OOM	
MedAR1	**0.8**	**173.3**	8.77	598.8			3.7	210.4^{2b}
MedAR2	33.2	**173.3**	15.7	599.4			**13.7**	210.4^{2b}
MedAR5	128.1	**173.3**	26.6	600			**12.9**	210.4^{2b}
MedHA1	**0.8**	**173.3**	8.9	598.7			5.5^{2c}	210.4^{2b}
MedHA2	33.2	**173.3**	14.7	599.5			**12.6**	210.4^{2b}
MedHA5	Timeout		3219.7	647.3			12.5	210.4^{2b}
MedHQ1	**0.7**	**173.3**	8.8	598.8			12.2	210.4^{2b}
MedHQ2	33.8	**173.3**	16.6	596.9			**13.2**	210.4^{2b}
MedHQ5	125.8	**173.3**	26.6	600			**12.6**	210.4^{2b}
Coverable instances								
Kanban	< **0.1**	**1.4**	804.7	55.1	Timeout		0.1	6.0^{2c}
pncsacover	2.8	**2.2**	7.9	11.2	36.5	8.8	**1.0**	23.0^{1c}
pncsasemiliv	0.1	**1.5**	0.2	3.9	32.1	8.8	< **0.1**	3.7^{2c}
MedAA1-bug	**0.8**	**172.7**	1.0	596.9	56.5	658.0	3.6	210.4^{2b}
MedHR2-bug	**0.6**	**172.7**	0.6	596.9	57.2	658.0	12.8	210.4^{2b}
MedHQ2-bug	0.4	**172.7**	**0.3**	596.9	56.8	658.0	12.9	210.4^{2b}

(MedXXX examples [18]). We focus on examples that took longer than 2 seconds for at least one algorithm. Table 1 show run times and memory usage on the mist2 and message-passing program benchmarks. For each row, the column in bold shows the winner (time or space) for each instance. IIC performs well on these benchmarks, both in time and in memory usage. To account for mist2's use of a pooled memory, we estimated its baseline usage to 2.5 MB by averaging over all examples that ran in less than 1 second. The memory statistics includes the size of the program binary. We created statically-linked versions of all binaries, and the binaries were within 1 MB of each other.

Multithreaded Program Benchmarks. Table 2 shows comparisons of IIC with MCOV on a set of multithreaded programs distributed with MCOV. While IIC is competitive on the uncoverable examples, MCOV performs much better on the coverable ones. We have identified two reasons for MCOV's better performance.

The first reason is an encoding problem. MCOV represents examples as thread transition systems (TTS) [16] that have 1-bounded "global" states and

Table 2. Experimental results: comparison between MCOV and IIC on examples derived from parameterized multithreaded programs. The superscripts for MCOV are as in Table 1.

Problem Instance	IIC Time	IIC Mem	MCOV Time	MCOV Mem
Coverable instances				
Boop 2	82.0	287.9	**0.1**	**12.1**^{1c}
FuncPtr3 1	< 0.1	**1.5**	< 0.1	3.4^{2c}
FuncPtr3 2	0.2	**12.3**	0.1	7.9^{2c}
FuncPtr3 3	28.5	939.1	**3.6**	**303.8**^{1c}
DoubleLock1 2	Timeout		**0.8**	**56.7**^{2c}
DoubleLock3 2	8.0	41.3	< **0.1**	**4.8**^{2c}
Lu-fig2 3	Timeout		**0.1**	**10.4**^{2c}
Peterson 2	Timeout		**0.2**	**23.0**^{1c}
Pthread5 3	132428	468.8	**0.1**	**17.0**^{1c}
Pthread5 3			0.2	**49.6**^{2c}
SimpleLoop 2	7.9	6.0	< **0.1**	**4.8**^{2c}
Spin2003 2	4852.2	54.4	< **0.1**	**2.7**^{2c}
StackCAS 2	2.5	**1.6**	< **0.1**	3.7^{2c}
StackCAS 3	5.5	21.7	< **0.1**	**4.4**^{2c}
Szymanski 2	Timeout		**0.4**	**26.7**^{2c}

Problem Instance	IIC Time	IIC Mem	MCOV Time	MCOV Mem
Uncoverable instances				
Conditionals 2	0.1	**3.6**	< **0.1**	5.7^{2c}
RandCAS 2	< **0.1**	**2.0**	< **0.1**	3.9^{2c}

potentially unbounded "local" states. A state of a TTS is a pair consisting of a single global state and a multiset of local states. While [16] defines multiple kinds of transitions, we only need to consider what happens to the global state. In TTS, each transition is of the form: given a global state g_1 and a condition on the local states, go to global state g_2 and modify the local states in a specific way. For example, the SimpleLoop-2 example is a TTS with 65 global states and 28 local states. In the TTS, we find that for each global state, there are at most 32 transitions that have any given global state as final global state. In particular, if the IIC algorithm were to be directly applied on this representation, the [Decide] rule could be applied at most 32 times to a given state generated by [Candidate], enumerating all pre-images induced by the corresponding transitions. Since our implementation works on Petri nets, we translate TTS to PN. The translation maps each global state to a Petri net place, but IIC does not know the invariant that exactly one of these places contains a token. Thus, IIC generates pre-images in which two or more places corresponding to global states contain tokens, and rules them out later through a conflict. On the translation of SimpleLoop-2, we found that 165 pre-images were generated for the target state in $R_1^{\downarrow}$. This causes a significant enlargement of the search space. We believe the use of PN invariants can improve the performance of IIC.

The second reason is the use of a combined forward and backward search in MCOV versus a backward search in IIC. It has been observed before that forward search performs better on software examples [13]. For comparison, we ran MCOV both in combined-search mode and in backward-search mode. In 80% of the cases (12 out of 16), the combined search was faster by at least a factor of 10, while no measurable difference was observed in the other four cases. Nevertheless, MCOV using backward search still outperforms IIC in these examples because of the better encoding.

In conclusion, based on experimental results, we believe that IIC, an IC3-based algorithm, is a practical coverability checker.

Acknowledgements. We thank Alexander Kaiser for help with the bfc tool. We thank Aaron Bradley for helpful and detailed comments and suggestions.

References

1. Abdulla, P.A., Cerans, K., Jonsson, B., Tsay, Y.-K.: General decidability theorems for infinite-state systems. In: LICS 1996, pp. 313–321. IEEE (1996)
2. Abdulla, P.A., Bouajjani, A., Jonsson, B.: On-the-fly analysis of systems with unbounded, lossy FIFO channels. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 305–318. Springer, Heidelberg (1998)
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
4. Ciardo, G.: Petri nets with marking-dependent arc multiplicity: properties and analysis. In: Valette, R. (ed.) ICATPN 1994. LNCS, vol. 815, pp. 179–198. Springer, Heidelberg (1994)
5. Cimatti, A., Griggio, A.: Software model checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
6. Dickson, L.E.: Finiteness of the odd perfect and primitive abundant numbers with n distinct prime factors. American Journal of Mathematics 35(4), 413–422 (1913)
7. Dufourd, C., Finkel, A., Schnoebelen, P.: Reset nets between decidability and undecidability. In: Larsen, K.G., Skyum, S., Winskel, G. (eds.) ICALP 1998. LNCS, vol. 1443, pp. 103–115. Springer, Heidelberg (1998)
8. Een, N., Mishchenko, A., Brayton, R.: Efficient implementation of property directed reachability. In: FMCAD 2011, pp. 125–134. FMCAD Inc. (2011)
9. Emerson, E.A., Namjoshi, K.S.: On model checking for non-deterministic infinite-state systems. In: LICS 1998, pp. 70–80. IEEE (1998)
10. Esparza, J., Finkel, A., Mayr, R.: On the verification of broadcast protocols. In: LICS 1999, pp. 352–359. IEEE Computer Society (1999)
11. Esparza, J., Nielsen, M.: Decidability issues for Petri nets – a survey. Bulletin of the EATCS 52, 244–262 (1994)
12. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! Theor. Comput. Sci. 256(1-2), 63–92 (2001)
13. Geeraerts, G., Raskin, J.-F., Van Begin, L.: Expand, enlarge and check: New algorithms for the coverability problem of WSTS. J. Comput. Syst. Sci. 72(1), 180–203 (2006)
14. Higman, G.: Ordering by divisibility in abstract algebras. Proceedings of the London Mathematical Society S3-2(1), 326–336 (1952)
15. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
16. Kaiser, A., Kroening, D., Wahl, T.: Efficient coverability analysis by proof minimization. In: Koutny, M., Ulidowski, I. (eds.) CONCUR 2012. LNCS, vol. 7454, pp. 500–515. Springer, Heidelberg (2012)
17. Kloos, J., Majumdar, R., Niksic, F., Piskac, R.: Incremental, inductive coverability. Technical Report 1301.7321, CoRR (2013)
18. Majumdar, R., Meyer, R., Wang, Z.: Static provenance verification for message-passing programs. In: SAS 2013 (2013)

Automatic Linearizability Proofs of Concurrent Objects with Cooperating Updates*

Cezara Drăgoi, Ashutosh Gupta, and Thomas A. Henzinger

Institute of Science and Technology Austria, Klosterneuburg

Abstract. An execution containing operations performing queries or updating a concurrent object is linearizable w.r.t an abstract implementation (called specification) iff for each operation, one can associate a point in time, called linearization point, such that the execution of the operations in the order of their linearization points can be reproduced by the specification. Finding linearization points is particularly difficult when they do not belong to the operations's actions. This paper addresses this challenge by introducing a new technique for rewriting the implementation of the concurrent object and its specification such that the new implementation preserves all executions of the original one, and its linearizability (w.r.t. the new specification) implies the linearizability of the original implementation (w.r.t. the original specification). The rewriting introduces additional combined methods to obtain a library with a simpler linearizability proof, i.e., a library whose operations contain their linearization points. We have implemented this technique in a prototype, which has been successfully applied to examples beyond the reach of current techniques, e.g., Stack Elimination and Fetch&Add.

1 Introduction

Linearizability is a standard correctness criterion for concurrent objects, which demands that all concurrent executions of the object operations are equivalent to sequential executions of some abstract implementation of the same object, called specification. In this paper, we address the problem of automatically proving linearizability for concurrent objects which store values from unbounded domains (such as counters and stacks of integers) and are accessed by an unbounded number of threads.

Concurrent objects (data structures) are implemented in wide audience libraries such as `java.util.concurrent` and Intel Threading Building Blocks. The search for new algorithms for concurrent data structures that maximize the degree of parallelism between the operations is an active research area. The algorithms have become increasingly complex, which makes proving their linearizability more difficult. Automatic verification techniques are expected not only to confirm the manual correctness proofs provided with the definition of the algorithms, but also to verify the various implementations of these algorithms in different programming languages.

Proving linearizability is a difficult problem because the configurations of the concurrent object and number of threads that access it are in general unbounded. An execution is *linearizable* if there exists a permutation of its *call* and *return* actions, preserving

* This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering) and by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 174–190, 2013.

the order between non-overlapping operations, which corresponds to a sequential execution allowed by the specification. This is equivalent to choosing for each operation an action between its *call* and *return*, called *linearization point*, such that the sequential composition of all operations, in the order in which the linearization points appear in the execution, belongs to the specification.

The techniques introduced in the literature for proving linearizability differ in the degree of automation and the class of libraries that they can handle. The works in [13,15] present semi-automatic techniques, where human guidance is required either to interact with the theorem prover [15] or to carry out mathematical reasoning. The work presented in [5] defines a class of concurrent objects for which linearizability is decidable if the object is accessed by a bounded number of threads. More related to the present paper, [2,3,17] present fully automatic techniques for proving linearizability based on abstract interpretation. Usually, automatic techniques prove linearizability using an abstract analysis of the concurrent implementation simultaneously with the sequential implementation of the abstract object. The analysis choses for each concurrent operation a linearization point. When the abstract execution reaches the linearization point of an operation, its sequential implementation is executed (with the same input as the concurrent one) and later, the returned values of the two operations are compared.

The concurrent objects which have been proven automatically linearizable by previous techniques have only operations with internal linearization points, i.e., when the linearization point of each operation is one of its actions, for all executions. A notable exception is [17], which lifts this restriction for operations whose specification doesn't update the abstract object. In this paper, we extend the state of art by introducing a technique to deal with concurrent objects whose *updating operations* have external linearization points. Several classical libraries such as [8,9,12,16] fall into this category.

When all operations have internal linearization points, there is a correspondence between an operation and its potential linearization points, which can be defined using assertions over variables of that operation and additional prophecy or ghost variables.

Defining such a correspondence for operations with external linearization points, i.e., the linearization point is an action of another, concurrently executing operation, is more difficult. In this case, for each action s one has to identify all possible concurrently executing operations whose linearization point could be s. Consequently, one needs an assertion language that allows relating the local variables of the operation executing s with the local variables of the threads executing the operations that have s as a linearization point, which is difficult to define and reason about.

This paper introduces a new technique for rewriting the implementation $\mathcal{L}$ of a concurrent object and its specification $\mathcal{S}$ into a new implementation $\mathcal{L}^n$ with a new specification $\mathcal{S}^n$, such that $\mathcal{L}^n$ preserves all executions of $\mathcal{L}$, and the linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$ implies the linearizability of $\mathcal{L}$ w.r.t. $\mathcal{S}$. The aim of the rewriting is that all operations of the new implementation contain their linearization points. Let us consider two operations o and a in execution e of $\mathcal{L}$, such that a contains the linearization point of o. The execution corresponding to e, obtained by rewriting $\mathcal{L}$, does not have o and a; instead it contains an operation of a new method, denoted $o+a$, which has the combined behavior of a and o, namely, $o+a$ has all the actions of o and a, in the same order in which they occur in the original execution. The rewriting consists in (1) adding new *combined*

methods, which are interleavings of methods that have external linearization points and methods that contain these linearization points, and (2) removing code fragments from methods in the original implementation whose behaviors are captured by the new, combined methods. The second step is crucial in order to eliminate operations with external linearization points. The specification of a combined method is the non-deterministic sequential composition of the method specifications whose interleaving it represents.

We have observed that an operation with external linearization points in concurrent object implementations such as [8,9,12,16] is included in a dependency cycle that witness the non-serializability of the execution in which it appears. Instead of searching for operations with external linearization points, we define an algorithm to compute an under-approximation of the set of dependency cycles and then, constructs combined methods whose executions correspond to interleavings that contain such cycles.

We reduce the problem of eliminating operations with external linearization points to the problem of eliminating non-serializable executions that contain dependency cycles. Concretely, the rewriting removes code fragments from the original implementation whose executions are included in the invocations of the combined methods. This step is reduced to checking some invariants over the executions of the original library, which are proven using a technique based on abstract interpretation [6] and counter abstraction [7]. Note that all these steps are automatic.

The linearizability of the original library is implied by a stronger version of linearizability for the new library, which restricts the possible choices for linearization points, to an interval strictly included in the time span of an operation, i.e., the interval of time between its call and return actions. Our technique rewrites operations having linearization points in other, concurrently-executing operations that form a dependency cycle identified by our algorithm. In our examples it eliminates all method invocations with external linearization points, and this allows us to use existing techniques [17,3] to prove the linearizability of the new implementation. A theoretical limitation of our approach is that it cannot deal with unbounded sets of operations that update the concurrent object and have the same linearization point. The rewriting procedure was implemented in a prototype, which has been successfully applied to examples beyond the reach of current techniques, e.g., Stack Elimination [9] and Fetch&Add [16].

In the following, Sec. 2 presents a motivating example; Sec. 3 characterizes executions with external linearization points; Sec. 4 defines the rewriting algorithm; Sec. 5 gives the correctness theorem for the rewriting and connects the linearizability of the new library with the linearizability of the original one.

2 Motivating Example

Example description: Let $\mathcal{L}_s$ be the library given in Fig. 1. It contains two methods, `push` and `pop`, which implement *Treiber's concurrent stack* extended with an elimination mechanism similar to the one used in the *Elimination stack* [9]. Each method is described by a control-flow graph (CFG, for short), whose nodes denote atomic blocks. We assume that each thread executes at most one operation, i.e., method instance.

The stack is implemented by a singly-linked list and `S` is a global pointer that points to the top of stack. In order to increase the degree of parallelism in Treiber's stack, which is limited by the sequential access to `S`, the elimination mechanism in [9] allows pairs

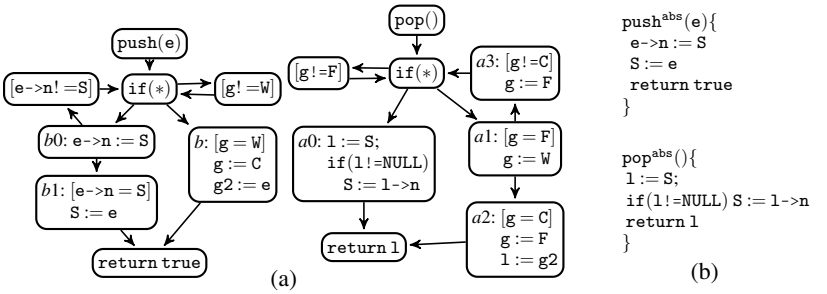


Fig. 1. (a) *Treiber's stack* with an elimination mechanism. The formulas in square brackets are assumes. Some code blocks are labelled for easy reference. (b) The specification is defined from the abstract implementations $\texttt{push}^{\texttt{abs}}$ and $\texttt{pop}^{\texttt{abs}}$ of push and pop.

of overlapping instances of pop and push to exchange their values without accessing S. Each method invocation non-deterministically chooses either to try to modify the stack or apply the elimination mechanism. The elimination mechanism is implemented as follows. If the value of g is W(ating), then there is an invocation of pop ready to exchange values and waiting for a push. If the value of g is C(ollided), then a pair of push and pop invocations is ready to exchange the value stored in g2. The exchange is complete when pop sets the value of g back to F(ree) so that another elimination can take place. The elimination mechanism is an example of cooperative update.

We define specifications with respect to *abstract implementations* of methods. The abstract implementations of push and pop are given in Fig. 1(b). For any method op, the abstract implementation $\texttt{op}^{abs}$ has the same arguments and return values. The specification $\mathcal{S}_s$ of $\mathcal{L}_s$ is the set of all sequential executions of the abstract implementations.

Linearizability: A (concurrent) execution e of a library $\mathcal{L}$ is linearizable w.r.t a specification $\mathcal{S}$ iff there is an execution s in $\mathcal{S}$ such that (1) for each operation in e there is an operation in s with the same interface (same invocation parameters and same responses) and (2) s preserves the order between non overlapping operations. The execution s is called the *linearization* of e. The library $\mathcal{L}$ is linearizable w.r.t $\mathcal{S}$ if all its executions are linearizable w.r.t $\mathcal{S}$. Fig. 2(a) shows an execution of $\mathcal{L}_s$ consisting of one instance of pop and two instances of push. Vertical dashed lines represent the context switches. Two possible linearizations of this execution are given in Fig. 2(b1) and (b2).

Linearization Points: Linearizability is often proved using the notion of linearization point [11]. An execution e of $\mathcal{L}$ is linearizable w.r.t. $\mathcal{S}$ iff for each operation op in e, one can choose an action of e located between the call and the return action of op, called linearization point, such that: the sequential composition of the abstract implementations corresponding to the operations in e, in the order defined by the linearization points, defines a linearization for e. The actions pointed to by dotted arrows in Fig. 2(a) are the linearization points that lead to the linearization in Fig. 2(b1) (resp., Fig. 2(b2)). The linearization points in Fig. 2 are called *internal* because they are actions of the operations whose linearization point they represent. A non-internal linearization point is called *external*. Proving automatically the linearizability of libraries with external linearization points is beyond the scope of all existing techniques we are aware of, except the work of Vafeiadis [17], which handles external linearization points but only

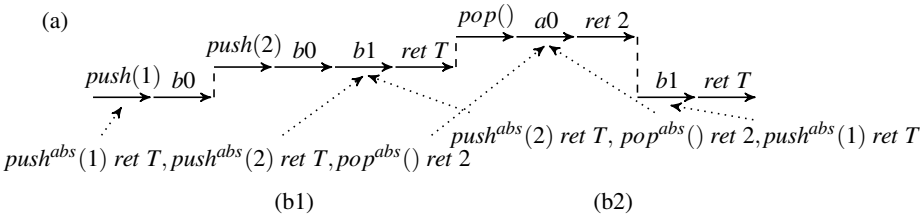


Fig. 2. A concurrent execution and two possible linearizations

for methods with specifications that *do not modify, but only read, the global data structure*.

Rewriting Libraries for Dealing with External Linearization Points: Let us consider the execution in Fig. 3, which consists of three operations $pop()$, $push(1)$, and $push(4)$ (a method name is written with typewriter font and its operations are written in italics). $pop()$ and $push(1)$ execute the elimination mechanism while $push(4)$ accesses the stack. Note that the linearization point of $pop()$ is *external* and it must be an action of $push(1)$. If we choose $a1$ as a linearization point for $pop()$, then there exists no sequential execution of the abstract implementations which exposes the same interface for all operations (intuitively, $pop()$ would have to return `Empty`). The same holds for all the other actions of $pop()$. A linearization for this execution is given in Fig. 3(a). If an operation contains the linearization point of another operation, e.g. $push(1)$ contains the linearization point of $pop()$, we say that the two instances *share linearization points*.

Our approach to proving linearizability of libraries with external linearization points (e.g. $\mathcal{L}_s$) is to rewrite the original library into a new library that has similar executions but a simpler linearizability proof (if any) w.r.t. a new specification, obtained from the specification of the original library.

> If the new library is linearizable w.r.t the new specification, then the original one is also linearizable w.r.t. its specification.

In the case of $\mathcal{L}_s$, our rewriting will introduce a new method `push+pop`, that will replace the elimination mechanism in $\mathcal{L}_s$. The execution of the new library corresponding to the execution in Fig. 3(a) is given in Fig. 3(b). The operations $pop()$ and $push(1)$, that exchange values via the elimination mechanism, have been rewritten into sub-sequences of an instance of a new method $push{+}pop(1)$. The time interval of $push{+}pop(1)$ is the union of the time intervals of $pop()$ and $push(1)$ and its interface is the union of the interfaces of $push(1)$ and $pop()$. The program instructions executed by $push{+}pop(1)$ are the same, and in the same order, as the ones in $push(1)$ and $pop()$, except for `call` and `return` instructions (the `call` and the `return` of $push(1)$ have been replaced by `skip`). The abstract implementation of `push+pop` corresponds to the sequential composition of $\texttt{push}^{abs}$ and $\texttt{pop}^{abs}$. A linearization for the new execution is given in Fig. 3(b). Notice that this linearization is defined using only internal linearization points.

The rewriting (1) adds new methods, called *combined methods*, that represent exactly those interleavings between the operations with external linearization points and the operations which contain these linearization points and (2) removes fragments of code from the methods of the original library such that these interleavings are not

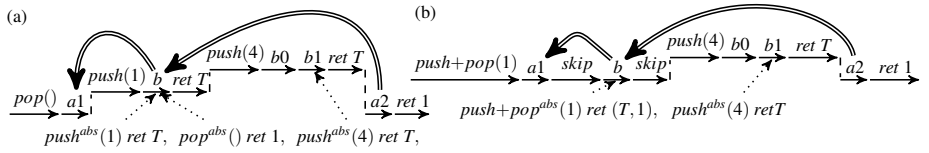


Fig. 3. (a) An execution of $\mathcal{L}_s$ where pop has only external linearization points. (b) An execution of $\mathcal{L}_s^n$ containing an invocation of the combined method push+pop.

possible anymore (they will be only instances of the combined methods). In the case of $\mathcal{L}_s$, to define the new library $\mathcal{L}_s^n$, we create combined methods, called push+pop, whose instances are interleavings of instances of pop and push that exchange values through the elimination mechanism. Also, we modify the code of pop, respectively push, in order to eliminate the instances executing the elimination mechanism. The specification of the new library is defined as follows: (1) the abstract implementation of each combined method is the non-deterministic sequential composition of the abstract implementations of the methods whose interleavings it represents and (2) the abstract implementation of the methods inherited from the original library remains the same. Fig. 5 shows one combined method from $\mathcal{L}_s^n$. Let $\mathcal{S}_s^n$ denote the new specification of $\mathcal{L}_s^n$.

In the case of $\mathcal{L}_s$, the new library has no operations with external linearization points and thus, its linearizability can be proved using the existing techniques.

Removing Retry Loops: The methods of a concurrent object often contain a retry loop, i.e., a loop where each iteration tries to execute the effect of the operation and if it fails it restarts forgetting any value computed in previous iterations. Given a library $\mathcal{L}$ whose methods contain retry loops, one can define a new library $\mathcal{L}'$ by replacing every retry loop `while` $(Cond)$ $\{Loop\_body\}$ with $Loop\_body$;`assume` *false* such that $\mathcal{L}$ is linearizable iff $\mathcal{L}'$ is linearizable. In principle, one can use classical data flow analyses in order to identify retry loops. In the following, we will consider concurrent objects implementations without loops.

3 Executions with External Linearization Points

In this section, we present a connection between the existence of operations with only external linearization points and serializability [14].

In an execution, a *conflict* is a pair of actions (program instructions) that access the same shared memory location and at least one of them modifies its value (each double arrow in Fig. 3 defines a conflict between its source and its destination). The *dependency graph* of an execution, is an oriented graph whose nodes represent operations and whose edges represent conflicts; the orientation is the order in which the actions in conflict appear in the execution. An execution is *serializable* (called also view-serializable) iff it can be reordered into an execution that has an acyclic dependency graph [1], called *serialization*, such that (1) the read actions read the same values as in the original execution

[1] The classical definition requires that the execution be sequential (a sequential composition of operations). However, any execution with an acyclic dependency graph can be reordered into a sequential execution with the same final state and where the read actions read the same values.

and (2) both executions have the same final state. The execution in Fig. 2(a) is serializable but the execution in Fig. 3(a) is not serializable. A conflict between a read and a write action such that the read follows the write in the execution is called a *data-flow dependency*. A cycle in the dependency graph such that all of its edges define data-flow dependencies is called a *data-flow dependency cycle*.

We make the following empirical observation relating executions with external linearization points and non-serializable executions:

> *If an execution contains two operations o and a s.t. the linearization point of o can only be an action of a and the abstract implementation of o modifies the logical state, then the execution is not serializable. Moreover, the two operations define a data-flow dependency cycle.*

Intuitively, the dependency cycle between o and a arises because the two operations communicate through global variables in both directions.

Communication from a to o: A linearization point can be thought of as the point in time where the logical effect of the method takes place and consequently, the value returned by a method should depend on the outcome of executing its linearization point. The action of a, which is the linearization point of o, is usually a write on a shared memory location, read later by o in order to determine its return value. In the execution from Fig. 3(a), $pop()$ returns the value written by $push(1)$ in variable `g2`.

Communication from o to a: Since o has a specification that modifies the logical state, o usually needs to communicate its intended modification to a via a shared memory location. In our example, $pop()$ assigns to `g` the value `W` in block $a1$ in order to announce to $push(1)$ that a pop is ready to exchange values.

In the following, we focus on data-flow dependency cycles s.t. any reordering of their actions does not lead to an execution of the library where they are not present anymore. These cycles were sufficient for all examples we have found in the literature.

4 Rewriting Algorithm

In this section, we describe the algorithm we propose for rewriting implementations of concurrent objects. The new implementations preserve all the behaviours of the original ones but, their executions contain fewer data-flow dependency cycles. To describe executions with data-flow dependency cycles we introduce a first order logic called $\mathbb{CL}$.

Let $\mathcal{L}$ be a library and $\mathcal{S}$ its specification. Let $[\![\mathcal{L}]\!]$ denote the set of executions of $\mathcal{L}$. The rewriting algorithm computes a library $\mathcal{L}^n$ and a specification $\mathcal{S}^n$ in several steps:

(1) Compute a set of formulas $\Lambda[\mathcal{L}]$ in $\mathbb{CL}$, where each formula in $\Lambda[\mathcal{L}]$ has a model in $[\![\mathcal{L}]\!]$ that has at least one data-flow dependency cycle (see Sec. 4.2).

(2) $\mathcal{L}^n$ is the union of $\mathcal{L}$ and a set of combined methods defined using the formulas in $\Lambda[\mathcal{L}]$. Any execution that contains an instance of a combined method corresponds to an execution in $[\![\mathcal{L}]\!]$, that satisfies some formula in $\Lambda[\mathcal{L}]$. The abstract implementation of each combined method is the non-deterministic sequential composition of the abstract implementations of the methods whose interleavings it represents (see Sec. 4.3).

(3) The methods of $\mathcal{L}^n$, that are copied from $\mathcal{L}$, are modified by removing CFG paths that are executed only in operations whose behaviour is captured by one of the

combined methods. The goal of this step is to make the formulas in $\Lambda[\mathcal{L}]$ unsatisfiable over the executions of $\mathcal{L}^n$ (see Sec. 4.4).

Note that $\Lambda[\mathcal{L}]$ captures a *subset* of the data-flow dependency cycles in $[\![\mathcal{L}]\!]$. Indeed, if more cycles are captured by $\Lambda[\mathcal{L}]$, then our rewriting will be more effective. For simplicity, the second and the third step of the algorithm are explained only for the case when $\Lambda[\mathcal{L}]$ is a singleton. Note that the rewriting detects and removes only sets of data-flow dependency cycles of finite length.

4.1 Logical Representations for Data-Flow Dependencies

The number of data-flow dependency cycles occurring in the executions of a library is potentially infinite. Therefore, we define a symbolic representation for sets of data-flow dependencies by formulas in a logic called *Conflict-cycles logic* ($\mathbb{CL}$, for short). This logic is parametrized by a library $\mathcal{L}$.

Formulas in $\mathbb{CL}$ define a partial order between pairs of conflicting actions and relations between the local states of the operations whose actions are in conflict. The variables of $\mathbb{CL}$ are interpreted over operations of $\mathcal{L}$. A model of a formula is an execution together with an interpretation of its variables to operations in this execution.

Let v, v_1, …, v_n denote variables of $\mathbb{CL}$ interpreted as operations. $\mathbb{CL}$ contains the following predicates and formulas:

– $v.a$ is a unary predicate, which holds iff in the considered execution, v interprets into an operation that executes the statement at control location a;

– $v_1.a < v_2.b$ is a binary predicate, which holds iff in the considered execution, the operation denoted by v_1 executes the statement at control location a before the operation denoted by v_2 executes the statement at control location b (we recall that, since we consider only loop-free methods, a location is reached only once by an operation);

– $v.a \rightarrow \mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathsf{g})$, is true iff the state in which v reached the control location a satisfies the boolean expression $\mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathsf{g})$, where $\mathit{Bool\_Expr}(v, v_1, \ldots, v_n, \mathsf{g})$ is build over the global variables g of $\mathcal{L}$ and the local variables of the threads executing $v, v_1, \ldots, v_n$.

$\mathbb{CL}$ formulas are conjunctions of the above predicates and formulas. For readability, the suffix of a variable name is a method name. Such a variable is interpreted only as an instance of that method, e.g., $v\mathtt{pop}$ is interpreted only as an instance of $\mathtt{pop}$. The *multiset of methods* in φ, denoted by $\mathsf{Op}(\varphi)$, consists of all methods whose instances are denoted by variables of φ. An execution e *satisfies* φ iff there is a mapping from the variables of φ to operations in e such that φ holds. $\mathcal{L}$ *satisfies* φ iff there is an execution of $\mathcal{L}$ that satisfies φ. A formula φ containing the atoms $v_0.a_0 < v_1.a_1, \ldots, v_{n-1}.a_{n-1} < v_n.a_n$ describes a dependency cycle if $v_0 = v_n$ and there is $i \in 0..n$ s.t. $v_i \neq v_0$. A model of such a formula φ has a data-flow dependency cycle if for each $i \in 1..n$ the actions corresponding to $v_{i-1}.a_{i-1}$ and $v_i.a_i$ form a data flow dependency.

Example 1. The formula $\varphi_{\mathcal{L}} = v\mathtt{pop}.a1 < u\mathtt{push}.b < v\mathtt{pop}.a2$ describes a dependency cycle. The execution in Fig. 3 is a model for $\varphi_{\mathcal{L}}$ where, $v\mathtt{pop}$ and $u\mathtt{push}$ are interpreted into $pop()$ and $push(1)$ respectively. The model has a data-flow dependency cycle, since $(pop().a1, push(1).b)$ and $(push(1).b, pop().a2)$ are data flow dependencies.

4.2 Under-Approximation of Data-Flow Dependency Cycles

To populate $\Lambda[\mathcal{L}]$, we search for executions containing finitely many concurrent operations that form a data-flow dependency cycle such that if one of the operations is removed from the interleaving, then the rest of them do not terminate.

We present an algorithm that given a library $\mathcal{L}$, computes an under-approximation of the data-flow dependency cycles exposed by executions of $\mathcal{L}$. It is parametrized by two *sequential* analyses: an abstract *reachability* analysis and a *may-alias* [2] analysis. To this, we define a recursive procedure $get\_cycle(l, \mathsf{op}, \mathsf{k})$ that receives as input a control location l from some method op of $\mathcal{L}$ and an integer k and returns a $\mathbb{CL}$ formula that represents data-flow dependency cycles of length at most k containing operations of op. This procedure works as follows:

(1) If l is reachable under the sequential analysis, following any of the paths in the CFG of op starting from the initial state of the library, denoted by S_0, then *get_cycle* returns *true*.

(2) Otherwise, let a be the first control location of an `assume` statement (on a path to l), which is infeasible. Using the may-alias analysis, *get_cycle* identifies a set of assignments Wr (in op or in other methods of $\mathcal{L}$) that may modify the variables in the `assume` statement.

(3) Choose an assignment in Wr at control location b of some method op' and, if $k > 0$ then recursively call *get_cycle* on b; otherwise return $false$. Let φ' be the formula returned by the recursive call.

$get\_cycle(ret, \mathtt{pop}, 2)$
/*1*/ $a2$ is unreachable
/*2*/ $Wr = \{b : g := C\}$
/*3*/ $\varphi' = get\_cycle(b, \mathtt{push}, 1)$
  /*1*/ b is unreachable
  /*2*/ $Wr = \{a1 : g := W\}$
  /*3*/ $\varphi' = get\_cycle(a1, \mathtt{pop}, 0)$
    /*1*/ $a1$ is reachable
    /*2*/ return $true$
  /*4*/ found interleaving: $\mathtt{pop}.a1;\mathtt{push}.b$
  /*5*/ return $\varphi' \wedge v\mathtt{pop}.a1 < u\mathtt{push}.b$
/*3*/ $Op(\varphi') = \{\mathtt{push}, \mathtt{pop}\}$
/*4*/ found interleaving:$\mathtt{pop}.a1;\mathtt{push}.b;\mathtt{pop}.a2$
/*5*/ return $v\mathtt{pop}.a1 < u\mathtt{push}.b < v\mathtt{pop}.a2$

Fig. 4. A run of *get_cycle* for computing data-flow dependency cycles in $\mathcal{L}_s$

(4) Search for an interleaving between (i) a path of op from its entry node to l, containing the failing `assume`, (ii) an interleaving of methods in $\mathsf{Op}(\varphi') \cup \{\mathsf{op}'\}$ s.t.
- the assignment of op' appears immediately before the `assume` of op,
- the interleaving of methods in $\mathsf{Op}(\varphi')$ satisfies the ordering constraints in φ',
- the `assume` of op is reachable (from S_0, under the considered reachability analysis).

If there is no such interleaving then, *get_cycle* returns *false*. If $\mathsf{op} \in \mathsf{Op}(\varphi')$ then, *get_cycle* begins by searching for an interleaving that contains only the methods in $\mathsf{Op}(\varphi')$ and satisfies all these constraints (this corresponds to the fact that an instance of op needed to make the assignment at b reachable is the same as the instance of op for which we want to prove that l is reachable). This is needed in order to complete a data-flow dependency cycle between the methods in $\mathsf{Op}(\varphi)$.

(5) If the control location l is reachable under the interleaving computed at (4) then, *get_cycle* returns $\varphi' \wedge (v\mathsf{op}'.b < v\mathsf{op}.a) \wedge (v\mathsf{op}'.b \rightarrow \psi)$, where ψ expresses the aliasing relating the variable assigned at b and the variables of the `assume` statement at a. Otherwise, it considers the next failing `assume` between a and l and goes to step 1.

[2] An analysis that determines if two variables may point to the same memory location.

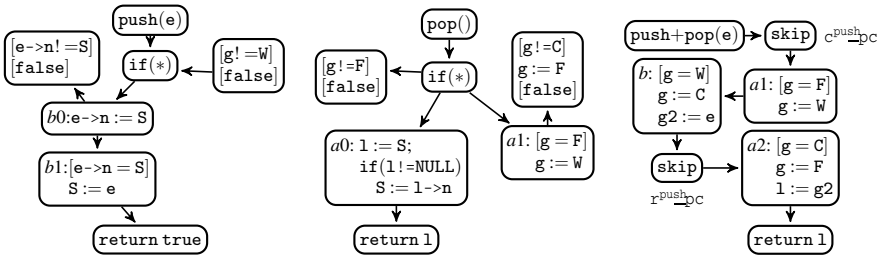


Fig. 5. Methods from the library $\mathcal{L}_s^n$ obtained by rewriting the library $\mathcal{L}_s$ in Fig. 1

Let $\Lambda[\mathcal{L}]$ be the union of the formulas generated by calling $get\_cycle(ret, \mathsf{op}, k)$ for each method op in $\mathcal{L}$, where *ret* denotes the return control location of op. To increase the precision, if $\Lambda[\mathcal{L}]$ contains two formulas $\varphi_1(v\mathsf{a}, v\mathsf{c})$ and $\varphi_2(v\mathsf{b}, v\mathsf{c})$ such that the actions of the method c appearing in both formulas lie on a common path in the CFG of c, we add $\varphi(v\mathsf{a}, v\mathsf{b}, v\mathsf{c}) ::= \varphi_1(v\mathsf{a}, v\mathsf{c}) \wedge \varphi_2(v\mathsf{b}, v\mathsf{c})$ to $\Lambda[\mathcal{L}]$ ($v\mathsf{a}, v\mathsf{b}, v\mathsf{c}$ are operations of the methods a, b, and c, respectively). This is necessary because the two formulas might describe two data-flow dependency cycles that share a method instance. Any execution satisfying a formula in $\Lambda[\mathcal{L}]$ contains data flow dependency cycles of length at most k. In Fig. 4, we present a run of *get_cycle* that returns $\varphi_{\mathcal{L}} = v\mathtt{pop}.a1 < u\mathtt{push}.b < v\mathtt{pop}.a2$.

4.3 Adding Combined Methods

Let φ be a formula in $\Lambda[\mathcal{L}]$ characterizing instances of a set of methods $\mathsf{op}_1, ..., \mathsf{op}_n$. We add to the new library a set of combined methods, denoted $\mathtt{Combined}(\varphi)$, which consists of all the interleavings between $\mathsf{op}_1, ..., \mathsf{op}_n$, that satisfy the ordering constraints in φ, instrumented by a set of assume statements that impose the relations between local variables expressed in φ. For each new method, all the call actions except for the first one and all the return actions except for the last one are replaced by skip. Also, all the accesses to the id of the thread that executes op_i, $i \in 1..n$, are replaced by accesses to a new local variable $\mathtt{tid}_i$ added to the combined method, which is initialized with a unique and random integer. The number of methods in $\mathtt{Combined}(\varphi)$ equals the number of total orders over the statements in $\mathsf{op}_1, ..., \mathsf{op}_n$ that access the global variables, which are consistent with the partial order defined by φ.

The input (resp., output) parameters of all methods in $\mathtt{Combined}(\varphi)$ are the union of the input (resp., output) parameters of $\mathsf{op}_1, ..., \mathsf{op}_n$. The abstract implementation of each combined method is a non-deterministic choice between all orders in which one can compose the abstract implementations of $\mathsf{op}_1, ..., \mathsf{op}_n$.

Example 2. In Fig. 5, we show the only method push+pop in $\mathtt{Combined}(\varphi_{\mathcal{L}})$, where $\varphi_{\mathcal{L}}$ is given in Ex. 1; the atomic blocks accessing global variables are totally ordered.

4.4 Removing Code Fragments from the Original Methods

The library obtained by adding combined methods still has executions with data-flow dependency cycles described by formulas in $\Lambda[\mathcal{L}]$. One needs to modify the code of the methods copied from $\mathcal{L}$ s.t. they do not generate instances satisfying formulas in $\Lambda[\mathcal{L}]$.

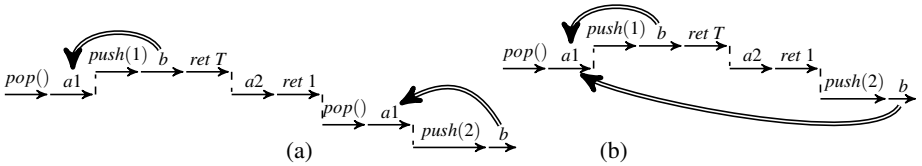


Fig. 6. (a) An execution of $\mathcal{L}_s$. (b) An execution that contains actions from push and pop but is not an execution of $\mathcal{L}_s$. Both executions are models of $Pref[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop})$ (double arrows emphasize the order relation between b and $a1$).

For simplicity, let us consider the case when $\Lambda[\mathcal{L}]$ contains only a formula φ. For any control location l in φ, $Pref[\varphi, l](v\texttt{op}, I)$ denotes the formula that describes the prefix of the interleavings characterized by φ, which end in l; $v\texttt{op}$ denotes the free variable of $Pref[\varphi, l]$ that is interpreted as the instance reaching l and I denotes all the other free variables. $Pref[\varphi, l](v\texttt{op}, I)$ contains all the predicates that constrain (local states at) control locations which are less than l in the partial order defined by the ordering constraints in φ and the program order.

Example 3. Let us consider the formula $\varphi_{\mathcal{L}}$ computed in Fig. 4 and the action associated with the atomic block b. Then, $Pref[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop}) = v\texttt{pop}.a1 < v\texttt{push}.b$, where $v\texttt{push}$ and $v\texttt{pop}$ denote an instance of the push, respectively pop, method. Note that $Pref[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop})$ captures only one direction of the communication between push and pop. Also, $Pref[\varphi_{\mathcal{L}}, a2](v\texttt{push}, v\texttt{pop}) = \varphi_{\mathcal{L}}$.

The original methods in $\mathcal{L}$ are modified by removing statements located at control locations appearing in φ. We remove those statements that are executed only in instances which can be generated by the combined methods. A control location l is removed from the CFG of the method op if the CFG contains a unique path starting in l and all the executions of $\mathcal{L}$ satisfy two invariants $\mathsf{Inv1}(l, \varphi)$ and $\mathsf{Inv2}(l, \varphi)$, defined in the following. We begin by explaining them on our running example.

In the continuation of Ex. 2, suppose that we want to eliminate the control location b from push. Intuitively, the invariant $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ states that, an instance of push reaches b iff there is an instance of pop that is ready to exchange values with the considered instance of push. Formally, in any execution, for any instance of push that reaches b there is a pop instance that reaches $a1$, which is expressed by the following formula:

$$\forall v\texttt{push}\ \exists v\texttt{pop}.\ v\texttt{push}.b \rightarrow v\texttt{pop}.a1 < v\texttt{push}.b \tag{1}$$

Intuitively, $\mathsf{Inv2}(b, \varphi_{\mathcal{L}})$ states that a pop exchanges values with at most one push; that is, in any execution, if there are two distinct instances of push that reach the atomic block b, then there are two distinct instances of pop such that each of them sends its job to a different push instance (when reaching $a1$). Formally, for any two distinct instances of push, denoted by $v\texttt{push}$, $v\texttt{push}'$, there are two distinct instances of pop, denoted by $v\texttt{pop}$, $v\texttt{pop}'$, such that $Pref[\varphi_{\mathcal{L}}, b](v\texttt{push}, v\texttt{pop}) \wedge Pref[\varphi_{\mathcal{L}}, b](v\texttt{push}', v\texttt{pop}')$ holds:

$$\forall v\texttt{push}, v\texttt{push}'\ \exists v\texttt{pop}, v\texttt{pop}'.\ \big(v\texttt{push} \neq v\texttt{push}' \wedge v\texttt{push}.b \wedge v\texttt{push}'.b\big) \rightarrow \\ \big(v\texttt{pop} \neq v\texttt{pop}' \wedge v\texttt{pop}.a1 < v\texttt{push}.b \wedge v\texttt{pop}'.a1 < v\texttt{push}'.b\big)$$

Any execution that satisfies these two properties can be rewritten such that any pair of instances of push and pop as above (reaching b and $a1$, respectively) becomes an

instance of some method `push+pop`. The execution in Fig. 6(a) satisfies both invariants while the execution in Fig. 6(b) satisfies only the first one.

These two invariants, together with the ones describing the conditions under which a_2 can be eliminated, state that one instance of `push` exchanges values with exactly one instance of `pop`. The code of `push` and `pop` in the new library is given in Fig. 5. The methods have the same names, but the atomic blocks b and $a2$ have been removed. Notice that, one cannot remove $a1$ because $a1$ has two successors. The algorithm has to preserve those instances where $a3$ is reached from $a1$.

A control location l is removed from the CFG of a method op of $\mathcal{L}$ iff its CFG contains a unique path starting in l and all executions of $\mathcal{L}$ satisfy the following invariants:

Inv1(l,φ)**:** any instance of op ending in l is a sub-sequence of an instance of one of the combined methods generated using φ. Since the combined methods are defined from $\mathcal{L}$ and φ, this invariant over the executions of $\mathcal{L}$ can be expressed as follows:

$$\forall v\mathsf{op}\ \exists I.\ v\mathsf{op}.l \rightarrow \mathit{Pref}[\varphi,l](v\mathsf{op},I)$$

i.e., for every instance $v\mathsf{op}$ of op there exist other method instances, denoted by I, such that if $v\mathsf{op}$ reaches the control location l, then the interleaving between $v\mathsf{op}$ and I is a prefix of one of the interleavings defined by φ;

Inv2(l,φ)**:** in any execution, each instance of op ending in l is a sub-sequence of a different instance of a combined method generated using φ:

$$\forall v\mathsf{op}, v\mathsf{op}'\ \exists I\ \exists I'.\ \begin{pmatrix} v\mathsf{op} \neq v\mathsf{op}' \wedge \\ v\mathsf{op}.l \wedge v\mathsf{op}'.l \end{pmatrix} \rightarrow \begin{pmatrix} \bigwedge_{x\in I, y\in I'} x \neq y \wedge \\ \mathit{Pref}[\varphi,l](v\mathsf{op},I) \wedge \mathit{Pref}[\varphi,l](v\mathsf{op}',I') \end{pmatrix}$$

To understand this invariant, suppose that it is not true and there exists a method instance that reaches l which is part of two distinct models of $\mathit{Pref}[\varphi,l]$. Note that it is not possible to represent these two models by two instances of a combined method (this holds because the instance that reaches l should be included in both).

Given a formula $\varphi \in \Lambda[\mathcal{L}]$, if none of the control locations l that appear in φ were removed from the code of the original methods, then all the methods in `Combined`(φ) are removed from the new library.

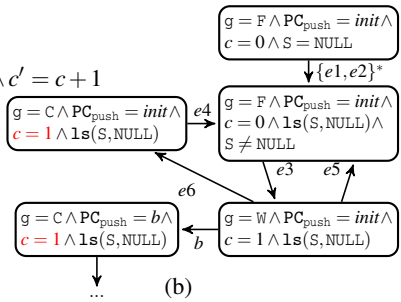
4.5 Invariant Checking

The invariants are *proved automatically* using a variation of the thread modular rely-guarantee analysis [10] over the concurrent system defined by the most general client of the library $\mathcal{L}$. The rely-guarantee analysis selects a statement of an arbitrary thread and executes it to generate *environment transitions*. The latter are obtained by existentially quantifying all local variables from the relation describing the effect of the selected statement. The environment transitions are subsequently applied to discover new transitions, until a fixed-point is reached.

For each invariant, our analysis needs to show that a given block of code executes only in a certain concurrent context. To compute a precise-enough over-approximation of the concurrent system, the analysis keeps a number of explicit copies of threads, called *reference threads*. All the other threads are called *environment threads*. The local states of the environment threads are described using a set of counters, each of them associated with a predicate over the local variables of an environment thread and the

$e1 ::= \ \mathtt{g}' = \mathtt{g} \wedge (\mathtt{ls}(\mathtt{S},\mathtt{NULL}) * \mathtt{S}' \mapsto \mathtt{S}) \wedge c' = c$
$e2 ::= \ \mathtt{g}' = \mathtt{g} \wedge (\mathtt{ls}(\mathtt{S}',\mathtt{NULL}) * \mathtt{S} \mapsto \mathtt{S}') \wedge c' = c$
$e3 ::= \ \mathtt{g} = \mathtt{F} \wedge \mathtt{g}' = \mathtt{W} \wedge \mathtt{g}2 = \mathtt{g}2' = \mathtt{NULL} \wedge \mathtt{Eq}(\mathtt{S},\mathtt{S}') \wedge c' = c+1$
$e4 ::= \ \mathtt{g} = \mathtt{C} \wedge \mathtt{g}' = \mathtt{F} \wedge \mathtt{g}2 \neq \mathtt{NULL} \wedge \mathtt{g}2' = \mathtt{NULL} \wedge \mathtt{Eq}(\mathtt{S},\mathtt{S}') \wedge c > 0 \wedge c' = c-1$
$e5 ::= \ \mathtt{g} = \mathtt{W} \wedge \mathtt{g}' = \mathtt{F} \wedge \mathtt{Eq}(\mathtt{S},\mathtt{S}') \wedge c > 0 \wedge c' = c-1$
$e6 ::= \ \mathtt{g} = \mathtt{W} \wedge \mathtt{g}' = \mathtt{C} \wedge \mathtt{g}2 = \mathtt{NULL} \wedge \mathtt{g}2' \neq \mathtt{NULL} \wedge \mathtt{Eq}(\mathtt{S},\mathtt{S}') \wedge c' = c$
$Env = \{e1, e2, e3, e4, e5, e6\}$

(a)



(b)

Fig. 7. (a) Abstraction of the concurrent system analyzed to prove *inv*1: $e1$ and $e2$ are the actions corresponding to pushing and popping an element in a stack, respectively; e3,e4,e5,e6 are the transitions over-approximating the elimination mechanism. (b) A part of abstract reachability graph with a reference thread executing `push`.

reference threads. These counters keep the number of environment threads that satisfy a certain predicate (over their local/global variables). The environment transitions refer to local variables of reference threads, global variables, and counters. Abstract states and environment transitions are represented by elements of a product between the abstract domains in [4] and a finite abstract domain $\{0, 1, +\infty\}$ describing the counters values.

The number of reference threads depends on the invariant, i.e., it is equal to the number of universally quantified variables in the invariant. The predicates are also derived from the invariant. For each invariant *inv*, we associate (1) a predicate $\mathtt{PC}{=}l$, for each control location l such that $v\mathtt{op}.l$ appears in *inv* and $v\mathtt{op}$ is existentially quantified, (this predicate holds iff the thread is at l) and (2) a predicate P, for each $v.a \rightarrow P$ an atomic formula in *inv* such that v is existentially quantified (the local variables of the universal instance variables are substituted by the local variables of the reference threads).

Example 4. To prove $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ in (1), we need to count how many threads are at the control location reached after executing $a1$. Let c be a counter associated with the predicate $\mathtt{PC}{=}a1$. Fig. 7(a) presents the set of environment transitions *Env* computed by our rely-guarantee analysis for $\mathcal{L}_s$ with counter c and one reference thread executing `push` (they are represented by formulas in Separation Logic containing, as usual, primed and unprimed variables). The predicate $\mathtt{ls}(\mathtt{S},\mathtt{NULL})$ denotes a singly linked list starting at `S` and ending in `NULL`, $\mathtt{S} \mapsto \mathtt{S}'$ states that the field `next` of `S` points to the memory cell pointed to by $\mathtt{S}'$; the macro $\mathtt{Eq}(\mathtt{S},\mathtt{S}')$ is used to say that the memory region reached from `S` did not changed. In Fig. 7(b), we present a part of the reachability graph obtained by executing *Env* in parallel with the reference thread executing `push`. On this reachability graph, we check that whenever the atomic block b is executed by the reference thread, the value of the counter c is greater or equal to one. Since $c = 1$, there is exactly one thread that executed the atomic block $a1$ from `pop`, so $\mathsf{Inv1}(b, \varphi_{\mathcal{L}})$ holds.

5 Correctness of the Rewriting Algorithm

In this section, we show the relation between the linearizability of the original library and the linearizability of the library obtained by applying the rewriting algorithm.

First, we show that the library $\mathcal{L}^n$, obtained by applying the rewriting algorithm on the library $\mathcal{L}$, preserves the behaviors of the original library, and that the specification $\mathcal{S}^n$ contains only sequential executions that are allowed by $\mathcal{S}$. That is, (i) every execution e of $\mathcal{L}$ can be rewritten into an execution e' of $\mathcal{L}^n$ s.t. interleavings of operations from e, which define *sets* of data-flow dependency cycles, are transformed in e' into instances of combined methods and (ii) every $s' \in \mathcal{S}^n$ is the rewriting of exactly one $s \in \mathcal{S}$.

Formally, an execution $e \in [\![\mathcal{L}]\!]$ is *similar* to $e' \in [\![\mathcal{L}^n]\!]$, denoted by $e \sim e'$, iff there is a function that transforms e into e' s.t. (1) there is a total function π between the threads of e' and sets of threads of e, (2) for each state and thread t of e', the set of thread-local states in $\pi(t)$ is aggregated into one thread-local state of t (the thread id's of $\pi(t)$ become local variables in the new state) and (3) the transformation preserves all actions of e, in the order they occur in e, except for some `call` and `return` actions, that are associated to `skip` actions. The first item states that for any execution e of the original library $\mathcal{L}$ there exists an execution e' of the new library $\mathcal{L}^n$ s.t. $e \sim e'$. This is denoted by $[\![\mathcal{L}]\!] \subseteq_{\sim} [\![\mathcal{L}^n]\!]$. The second item states that, for any $s' \in \mathcal{S}^n$, there exists exactly one $s \in \mathcal{S}$ s.t. $s' \sim^{-1} s$, where $\sim^{-1}$ is the inverse of $\sim$. This is denoted by $\mathcal{S}^n \subseteq^{!}_{\sim^{-1}} \mathcal{S}$.

Example 5. The execution e given in Fig. 3(a) is similar to the execution e' given in Fig. 3(b), i.e., $e \sim e'$. Also, the linearization s in Fig. 3(a) is similar to the linearization s' in Fig. 3(b), i.e., $s' \sim^{-1} s$.

Theorem 1. *Let $\mathcal{L}^n$ and $\mathcal{S}^n$ be the output of the rewriting algorithm for the input library $\mathcal{L}$ and its specification $\mathcal{S}$. Then,*

$$[\![\mathcal{L}]\!] \subseteq_{\sim} [\![\mathcal{L}^n]\!] \text{ and } \mathcal{S}^n \subseteq^{!}_{\sim^{-1}} \mathcal{S}.$$

Theorem 1 follows from the following lemma, which states the necessary conditions for removing statements from the CFG of a method in $\mathcal{L}$.

Lemma 1. *Let $\mathcal{L}$ be a library, φ a $\mathbb{CL}$ formula, and l a control location in a method* `op` *of $\mathcal{L}$ such that there is a unique path in the CFG starting from l. If $Inv_1(l,\varphi)$ and $Inv_2(l,\varphi)$ hold for all executions of the most general client of $\mathcal{L}$, then any execution e in $[\![\mathcal{L}]\!]$ is similar to an execution e' in $[\![\mathcal{L}^n]\!]$ such that all the operations from e reaching l correspond in e' to sub-sequences of instances of methods in* `Combined`(φ).

We introduce a stronger version of linearizability, called *R-linearizability*, and show that the *R*-linearizability of the new library $\mathcal{L}^n$, generated by the rewriting algorithm, w.r.t. $\mathcal{S}^n$ implies the linearizability of the original library $\mathcal{L}$ w.r.t. $\mathcal{S}$. One needs to prove the *R*-linearizability of the new library, instead of classical linearizability, because the induced rewriting on executions forgets ordering constraints between non-overlapping operations, more precisely, between pairs of operations such that at least one of them is rewritten (together with other operations) into a combined operation.

For example, let e' be the execution from Fig. 3(b) of the concurrent stack $\mathcal{L}^n_s$ given in Fig. 5. Because the instances of `push` + `pop` and `push` are overlapping, e' has two correct linearizations w.r.t. $\mathcal{S}^n_s$, i.e.,

$$s'_1 = push + pop(1)ret(1)\ push(4)\ ret(T) \text{ and } s'_2 = push(4)\ ret(T)\ push + pop(1)ret(1).$$

The only executions similar to s'_1 and s'_2 in the specification $\mathcal{S}$ (see Th. 1) are

$$s_1 = push(1)ret(T)pop()ret(1)push(4)\ ret(T),\ s_2 = push(4)\ ret(T)push(1)ret(T)pop()ret(1),$$

respectively. We recall that the execution e in Fig. 3(a) is similar to e'. Notice that, the classical linearizability of e' is not sufficient to imply the linearizability of e, because not every linearization of e' leads to a linearization of e. For example, s_2, which is similar to s'_2 is not a correct linearization of e because the two operations of `push` in e are non-overlapping and the order between them is not respected by s_2.

To overcome this problem, the rewriting distinguishes for each combined method `op` two control locations: $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ is the `skip` statement replacing the last `call` action and $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$ is the `skip` statement replacing the first `return` action in the interleaving represented by `op` (Fig. 5 distinguishes the control locations $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$ for the method $\mathtt{push} + \mathtt{pop}$). For the methods inherited from the original library, $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$ are the locations of their `call` and `return` actions. Then, instead of showing the linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$, we show that for any execution e' in $[\![\mathcal{L}^n]\!]$ there exists a sequential execution s' in $\mathcal{S}^n$ s.t. 1) for each operation in e' there is an operation in s' with the same interface and (2) s' preserves the order between non overlapping sequences of actions of the same method starting at $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ and ending at $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$.

Formally, let $\mathcal{L}$ be a library and `Rcp` a mapping that associates to each method `op` of $\mathcal{L}$ two distinguished statements which are not on a loop, denoted $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$. An execution e of $\mathcal{L}$ is *R-linearizable w.r.t. $\mathcal{S}$ and* `Rcp` iff e is linearizable w.r.t $\mathcal{S}$ and for each invocation *op* of `op` in e, one can choose a linearization point located between the execution of the actions associated with $\mathtt{c}^{\mathtt{op}}\mathtt{pc}$ and $\mathtt{r}^{\mathtt{op}}\mathtt{pc}$. The sequential execution in $\mathcal{S}$ corresponding to this sequence of linearization points is called the *R-linearization* of e. When `Rcp` contains only the `call` and `return` actions of each method, *R*-linearizability coincides with the classical definition of linearizability. The rewriting algorithm identifies a mapping `Rcp` for the new library such that for any execution e of $\mathcal{L}$, if the execution e' in $[\![\mathcal{L}^n]\!]$ to which it is similar, i.e., $e \sim e'$, is *R*-linearizable, i.e. there exists s' an *R*-linearization of e', then any $s \in \mathcal{S}$ such that $s \sim s'$ is a correct linearization of e.

For example, the *R*-linearization of the execution e' in Fig. 3(b) w.r.t. the mapping `Rcp` defined as above is s'_1. Thus, s_1 is a linearization of the execution e in Fig. 3(a).

Theorem 2. *Let $\mathcal{L}^n$ and $\mathcal{S}^n$ be the library, resp. the specification, obtained by rewriting a library $\mathcal{L}$ and its specification $\mathcal{S}$. The R-linearizability of $\mathcal{L}^n$ w.r.t. $\mathcal{S}^n$ and the mapping* `Rcp` *defined by the rewriting algorithm implies the linearizability of $\mathcal{L}$ w.r.t. $\mathcal{S}$.*

6 Experimental Results

The crucial steps of the rewriting, generating the set of formulas $\Lambda[\mathcal{L}]$ representing dependency cycles and checking the invariants in Sec. 4.4, are implemented into a prototype tool, which is based on the abstract domain in [1]. Table 1 presents a summary of the obtained results [3]. For each of the considered libraries, $\#op_{\mathcal{L}}$, resp. $\#op_{\mathcal{L}^n}$, is the number of methods in the original library, resp. the new library, #*cycles* is the number of cycles discovered by *get_cycle*, #*pcycles* is the number of cycles, among the ones discovered by *get_cycle*, for which the abstract concurrent analysis proved the required invariants, and #*pc* is the number of control points removed from the original methods.

[3] More details are available at `http://pub.ist.ac.at/~cezarad/lin2lin.html`

Table 1. Experimental results on an Intel Core i3 2.4 GHz with 2GB of memory

library $\mathcal{L}$	*size of* $\mathcal{L}$ #op	*get_cycle* k	#cycles	time(sec)	*inv check* #pcycles	time(sec)	#pc	size of $\mathcal{L}^n$ #op
`Simple Stack Elim.`	2	2	1	<1	1	<1	2	3
`Stack Elim.`	1	2	2	<3	2	<3	6	8
`Modified Stack Elim.`	2	2	2	<3	2	<3	6	9
`Fetch&Add`	1	3	4	<3	4	<4	3	8

The implementation of Fetch&Add allows an unbounded number of operations to have the same linearization point. We have analyzed a modified version of this implementation, that introduces a bound on the number of operations whose task can be performed by the same concurrently executing operation. Similarly, the implementations based on "flat combining" [8] are tractable by our method if we bound the number of threads that make public the signature of their updates in a given time interval. This restriction is natural when implementing concurrent objects in a real programing language because otherwise, the library has runs without progress on the data structure.

All the libraries we obtained after the rewriting are *R*-linearizable. Furthermore, all their methods have *internal linearization points* (e.g., the action associated with *b* is the linearization point of `push` + `pop` in Fig. 5), which allows us to prove their *R*-linearizability using existing techniques, e.g., [3,17]. The only exception is Queue Elimination [12] which is not *R*-linearizable. However, the library obtained by rewriting Queue Elimination is linearizable and this implies the linearizability of the original library because of some closure properties of the queue specification.

References

1. CINV, `http://www.liafa.univ-paris-diderot.fr/cinv/`
2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearizability. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 477–490. Springer, Heidelberg (2007)
3. Berdine, J., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
4. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: On inter-procedural analysis of programs with lists and data. In: Proc. of PLDI, pp. 578–589 (2011)
5. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent list implementations. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 465–479. Springer, Heidelberg (2010)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: Proc. of POPL, pp. 238–252 (1977)
7. German, S.M., Sistla, A.P.: Reasoning about systems with many processes. J. ACM 39(3), 675–735 (1992)
8. Hendler, D., Incze, I., Shavit, N., Tzafrir, M.: Flat combining and the synchronization-parallelism tradeoff. In: Proc. of SPAA, pp. 355–364 (2010)
9. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: Proc. of SPAA, pp. 206–215 (2004)

10. Henzinger, T.A., Jhala, R., Majumdar, R., Qadeer, S.: Thread-modular abstraction. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 262–274. Springer, Heidelberg (2003)
11. Herlihy, M., Wing, J.M.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
12. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free fifo queues. In: Proc. of SPAA, pp. 253–262 (2005)
13. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: Proc. of PODC, pp. 85–94 (2010)
14. Papadimitriou, C.H.: The serializability of concurrent database updates. J. ACM 26(4), 631–653 (1979)
15. Schellhorn, G., Wehrheim, H., Derrick, J.: How to prove algorithms linearisable. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 243–259. Springer, Heidelberg (2012)
16. Shavit, N., Zemach, A.: Combining funnels: A dynamic approach to software combining. J. Parallel Distrib. Comput. 60(11) (2000)
17. Vafeiadis, V.: Automatically proving linearizability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 450–464. Springer, Heidelberg (2010)

DUET: Static Analysis for Unbounded Parallelism

Azadeh Farzan and Zachary Kincaid

University of Toronto

Abstract. DUET is a static analysis tool for concurrent programs in which the number of executing threads is not statically bounded. DUET has a modular architecture, which is based on separating the invariant synthesis problem in two subtasks: (1) data dependence analysis, which is used to construct a data flow model of the program, and (2) interpretation of the data flow model over a (possibly infinite) abstract domain, which generates invariants. This separation of concerns allows researchers working on data dependence analysis and abstract domains to combine their efforts toward solving the challenging problem of static analysis for unbounded concurrency. In this paper, we discuss the architecture of DUET as well as two data dependence analyses that have been implemented in the tool.

1 Introduction

Verification of concurrent programs is a notoriously challenging problem. The difficulty arises from the fact that the size of the control space of a concurrent program is exponential in the number of executing threads, which makes direct analysis of the control space infeasible. The problem is even more difficult for programs where the number of executing threads is not bounded: in this case, the control space is *infinite*.

In this paper, we present DUET, a static analysis tool for analyzing programs with *unbounded parallelism*. DUET is based on the general philosophy that *general invariant generation can be reduced to data-dependence analysis*. This philosophy is particularly interesting in the case of programs with unbounded parallelism, since in this setting invariant generation is a formidable problem but data-dependence analysis is tractable. In this paper, we expound on the philosophy of DUET. We describe its architecture and describe two different analyses implemented in the tool [4,5]. Finally, we present experimental results demonstrating the efficacy of our tool on a set device drivers.

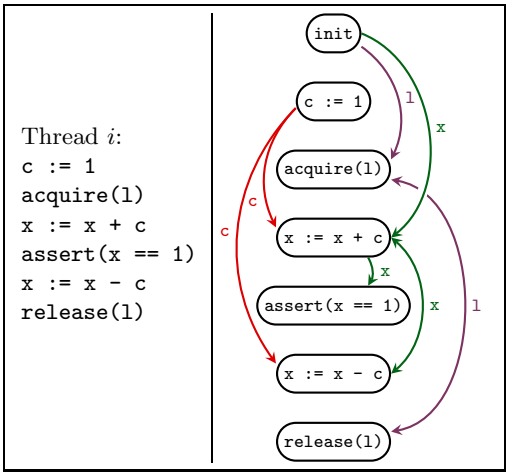
2 Overview

DUET is based on the idea that much of the essential behaviour of a program is captured by the *data flow* of the program. Consider the program below, in which unboundedly many copies of the thread Thread i execute in parallel. A *data flow graph* (DFG) for this program is pictured to its right. The edges of this DFG match each read of a variable with the writes that *may* reach it: for example, the edge from `c := 1` to `x := x + c` labeled `c` indicates that the value of `c` may

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 191–196, 2013.

flow from `c := 1` to `x := x + c` (and thus, `c` *may* be `1` at `x := x + c`). Note that some edges in this graph go in the opposite direction of control flow (e.g., the double-headed edge between `x := x + c` and `x := x - c`, which indicates a pair of edges, one in each direction). The value of `x` may flow from `x := x - c` to `x := x + c` because these instructions may be executed by different threads. Notice also that several interesting edges *do not* exist: there is no `x`-labeled loop on `x := x - c`; Lock `l` enforces the atomicity of the lock block, which breaks this data flow; an occurrence of `x := x + c` must always be executed between any two `x := x - c` instructions.

A data flow graph represents a system of constraints that can be solved automatically (using standard techniques) to yield program invariants [5]. This is the essential idea of the Duet tool: since DFGs can be used to compute invariants, invariant generation can be reduced to constructing a DFG for a program. Moreover, since data flow captures an essential feature of program behaviour, this strategy is often sufficient to prove properties of interest. For the example program above, Duet is able to prove that assertion always holds by interpreting the DFG over an interval abstract domain.



3 Duet's Architecture

The high-level architecture of Duet is pictured in Figure 1. We will begin by describing the basic work-flow of Duet and then discuss each of the component modules in more detail.

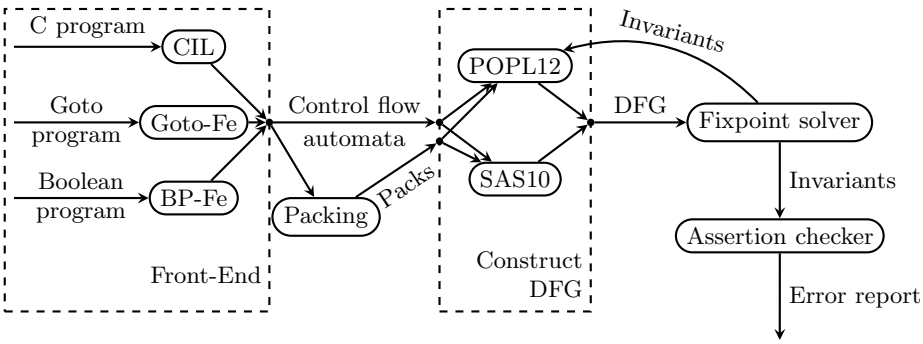


Fig. 1. Duet's architecture

First, one of the three **front-end** modules (CIL, Goto-Fe, BP-Fe) takes a (C, Goto, or Boolean) program as input and produces a system of control flow automata, with one automaton for each thread of the program. Next, a variable **packing** algorithm determines a set of *packs* (a *pack* is a set of semantically related variables, according to some heuristic). The set of packs and the system of control flow automata are fed into one of the two **construct DFG** modules (POPL12 or SAS10), which uses a data dependence analysis to construct a DFG for the program. The **fixpoint solver** interprets the DFG over some abstract domain to compute a set of invariants, which are passed to the assertion checker (in the case of POPL12, the invariants may alternatively be fed back into the DFG construction module). The **assertion checker** uses the invariants to determine check which assertions are safe and which *may* fail, and generates an error report.

3.1 Front-End

DUET accepts three types of inputs: (1) C programs using *pthreads* library for thread operations (using the CIL front-end [9]), (2) Boolean programs[1], or (3) goto programs, as produced by the GOTO-CC C/C++ front-end (part of the CPROVER project [1]). The front-end transforms these programs into a common form, namely a system of control flow automata. It also annotates the program with assertions, according to user input (for example, an assertion `assert(p != 0)` is generated for each access path of the form `*p`, if the `-check-null-pointer` option is set).

3.2 Variable Packing

In the DFGs implemented in DUET, each edge is labeled by a *pack* rather than a variable. A pack is a set of variables that may be related to one another. We may think of a DFG edge labeled by a pack as "carrying" a value for each variable in that pack (which allows an abstract domain to correlate variables belonging to a pack). The case where edges are labeled by variables is the special case where all packs are singleton sets. The **packing** module computes a set of packs from an input program, and passes those packs to the DFG construction phase. The choice of the abstract domain (non-relational vs relational) determines the choice of the packing algorithm used (there are currently two options in DUET).

3.3 DFG Construction

A DFG construction module takes a system of control flow automata and a set of packs (and in the case of POPL12, a set of invariants), and constructs a DFG representation of the program. There are currently two strategies for constructing DFGs that are implemented in DUET, which we describe below.

Nested locks (SAS10). In [4], we leverage reachability results for concurrent programs communicating via nested locks [6] to develop a compositional technique for solving bitvector analysis problems. Data-dependence analysis can be formulated as a bitvector problem, so we may use this technique to construct DFGs. Our method computes a summary for each thread describing its behaviour and then

[1] `http://www.cprover.org/boolean-programs/`

composes the summaries to compute a DFG for the program. This compositional approach enables our analysis to be sound and precise (it computes meet-over-feasible-paths solutions) even when the number of executing threads in unbounded.

Global variables (POPL12). A more challenging (and also strictly more general) program model uses global variables for synchronization rather than locks. This model is difficult because it requires circular reasoning: in order to perform a reasonably accurate data dependence analysis to construct a DFG we need invariants for the synchronization variables, and in order to compute invariants for the synchronization variables we need to construct a DFG.

In [5], we present an approach based placing the DFG construction module (**POPL12**) and the **fixpoint solver** into a feedback loop. The DFG and the invariants are iteratively coarsened as the algorithm progresses (that is, the invariants become weaker, and more edges are added to the DFG) until a fixpoint is reached. When a fixpoint is reached, the invariants overapproximate the dynamic behaviour of the program in question.

3.4 Fixpoint Solver

The **fixpoint solver** module interprets a DFG over an abstract domain to yield program invariants. It accomplishes this by computing a weak topological order on the DFG and repeatedly evaluating DFG nodes over the chosen abstract domain until a fixpoint is reached, as in [3]. Duet employs the abstract domains implemented in Apron [2] for this task.

3.5 Assertion Checker

Finally, the **assertion checker** module iterates over the DFG vertices: for each assertion vertex, we determine whether the invariant at that location (computed by the fixpoint solver) implies the assertion using the Apron [2] library. If the check fails, the assertion *may* fail, and we add it to the error report.

4 Experiments

We used a benchmark suite of 15 Linux device drivers to evaluate Duet. Since a driver may have arbitrarily many clients, these programs exhibit unbounded parallelism. Table 1 presents the result of running Duet on a collection of 15 Linux device drivers. These drivers are all written in C, and include infinite data (such as integer types).

With an interval analysis, Duet manages to prove most of the assertions correct (1312 out of a total 1597), and does so in 13 minutes. Duet's performance using an octagon analysis is slightly worse, proving 1277 assertions correct in 90 minutes.

Most false positives for Duet appear to be caused by one of two reasons: imprecision in the abstract domain (e.g. lack of a precise enough abstraction to handle zero-terminated arrays that are used in most of these drivers), and imprecision in how Duet handles the treatment of spinlocks in goto programs (due to the imprecision in the alias analysis that for lock variables). Neither of

Table 1. DUET's Performance on Integer Programs with unbounded parallelism, run on an 3.16GHz Intel(R) Core 2(TM) machine with 4GB of RAM

Device Drivers	#assertions	DUET: Interval Analysis		DUET: Octagon Analysis	
		safe	time	safe	time
i8xx_tco	90	75	1m51s	71	1m25s
ib700wdt	75	64	30s	64	20s
machzwd	87	73	39s	67	14m44s
mixcomwd	91	72	22s	74	25
pcwd	240	147	2m43s	145	23m48s
pcwd_pci	204	187	2m18s	188	2m59s
sbc60xxwdt	91	77	28s	69	11m27s
sc520_wdt	85	71	28s	65	13m20s
sc1200wdt	77	66	34s	66	33s
smsc37b787_wdt	93	80	47s	80	47s
w83877f_wdt	92	78	29s	72	13m24s
w83977f_wdt	101	90	34s	82	34s
wdt	99	88	25s	86	25s
wdt977	88	77	27s	75	28s
wdt_pci	84	67	33s	66	5m33s
total	1597	1312	13m9s	1270	90m21s

these sources of imprecision is due to a fundamental limitation of the analysis technique proposed in this paper (or related to concurrency). But, they hint on the idea that more precise alias analysis techniques (for concurrent code) and better abstract domains could hugely benefit the false positive rate of DUET.

Interval vs Octagon Analysis. The table on the right compares the results of interval and octagon analyses in DUET over the driver benchmarks (same experiments as in Table 1. Octagon analysis performs worse than interval analysis, however, it manages to prove 3 assertions safe that interval analysis fails. Octagon analysis is very sensitive to the variable packing algorithm. Investigating more sophisticated packing algorithms is a topic of our future work.

		OCT	
		safe	unsafe
IVL	safe	1267	45
	unsafe	3	282

4.1 Boolean Programs

Although Boolean programs are not the intended target of DUET, we performed a set of experiments over existing Boolean program benchmarks for two reasons: (1) to compare with two recent approaches [8,7] for verification of concurrent Boolean programs with unboundedly many threads, and (2) there is no aliasing present in Boolean programs, which limits the scope of implementation-related imprecision for a better evaluation of the core method.

Even though DUET can directly analyze the original device driver codes (i.e. does not require a predicate abstraction phase), we chose to compare with the existing tools [8,7] on the Boolean abstractions (for which these tools were designed), to present a more fair comparison. We compare DUET against two recent algorithms that handle Boolean programs with unbounded parallelism: dynamic cutoff detection (DCD) from [7], as implemented in Boom, and linear interfaces (LI) from [8],

as implemented in Getafix. We compared these tools against the benchmarks provided by the authors. It is important to note that both tools are capable of finding counterexamples to definitively declare an assertion unsafe, whereas, when DUET fails to produce strong enough invariants to prove an assertion correct, it is not clear whether the assertion is incorrect or DUET failed to prove it correct. The timeout is 5 minutes in all cases.

		LI			
		safe	unsafe	timeout	unknown
DUET	safe	1289	0	267	957
	unknown	54	247	23	579
	timeout	0	0	0	0

The table on the left presents the results of comparing DUET against LI over LI benchmarks. Columns/ rows titled safe, unsafe, and timeout are self-explanatory. An "unknown" response from DUET means that the invariants were not strong enough to prove the assertion safe, and an "unknown" response from LI means that neither a counter example was found nor the program was proved safe. DUET substantially outperforms LI; for example, there were 957 assertions proved safe by DUET that were declared unknown by LI, where as LI could only prove safe 54 of DUET's unknown cases.

The table on the right presents the results of comparing against DCD over DCD benchmarks. Again, DUET substantially outperforms DCD in proving assertions correct; 39 more assertions (out of 58) are proved correct by DUET.

		DCD		
		safe	unsafe	timeout
DUET	safe	19	0	39
	unknown	0	203	2
	timeout	0	7	0

References

1. Alglave, J., Kroening, D., He, N., Ranjan, A., Seghir, N., Tautschnig, M.: CPROVER Project (November 2011)
2. Jeannet, B., Miné, A.: Apron: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
3. Bourdoncle, F.: Efficient chaotic iteration strategies with widenings. In: Pottosin, I.V., Bjorner, D., Broy, M. (eds.) FMP&TA 1993. LNCS, vol. 735, pp. 128–141. Springer, Heidelberg (1993)
4. Farzan, A., Kincaid, Z.: Compositional bitvector analysis for concurrent programs with nested locks. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 253–270. Springer, Heidelberg (2010)
5. Farzan, A., Kincaid, Z.: Verification of parameterized concurrent programs by modular reasoning about data and control. In: POPL, pp. 297–308 (2012)
6. Kahlon, V., Ivančić, F., Gupta, A.: Reasoning about threads communicating via locks. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 505–518. Springer, Heidelberg (2005)
7. Kaiser, A., Kroening, D., Wahl, T.: Dynamic cutoff detection in parameterized concurrent programs. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 645–659. Springer, Heidelberg (2010)
8. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking parameterized concurrent programs using linear interfaces. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 629–644. Springer, Heidelberg (2010)
9. Necula, G.C., McPeak, S., Rahul, S.P., Weimer, W.: Cil: Intermediate language and tools for analysis and transformation of c programs. In: CC, pp. 213–228 (2002)

SVA and PSL Local Variables - A Practical Approach

Roy Armoni, Dana Fisman, and Naiyong Jin

Synopsys

Abstract. SystemVerilog Assertions (SVA), as well as Property Specification Language (PSL) are linear temporal logics based on LTL [14], extended with regular expressions and local variables. In [6] Bustan and Havlicek show that the local variable extensions, as well as regular expressions with intersection, render the verification problem of SVA and PSL formulae EXPSPACE-complete. In this paper we show a practical approach for the verification problem of SVA and PSL with local variables. We show that in practice, for a significant and meaningful subsets of those languages, which we denote PSL^{pract}, local variables do not increase the complexity of their verification problem, keeping it in PSPACE.

1 Introduction

SystemVerilog is an industrial standard unified hardware design and verification language. SystemVerilog Assertions (SVA) [11,7] is a subclass of SystemVerilog, used to declaratively specify functional behaviours of hardware designs. Similarly, Property Specification Language (PSL) [10,8] is used to declaratively specify functional behaviours of hardware designs independent of the design language. Typically, both SVA and PSL properties are then either validated during dynamic simulation or formally verified.

Several works during the recent decade defined industrial functional specification languages and studied the complexity of the verification problem of those languages [2,1,4,6].[1] SVA, part of the IEEE 1800 SystemVerilog standard, is a linear time temporal logic, based on Pnueli's LTL [14], extended with syntactic sugaring, regular expressions with intersection, connectives between regular expressions and formulae, and local variables. PSL, IEEE 1850 standard, has similar constructs except that its local variables' semantics differs slightly [9].

The expressiveness of SVA and PSL as well as the worst case complexity of their verification problem have already been studied before. Armoni et al [1] show that the expressive power of LTL extended with regular expressions (the ForSpec temporal logic) is exactly all the ω-regular languages. They also show that the complexity of the verification problem for this language is PSPACE-complete in the absence of time windows, and becomes EXPSPACE-complete when time windows are present. Bustan and Havlicek [6] further investigate the complexity of SVA verification with four independent extensions, namely intersection of regular expressions, local variables, PSL flavoured quantified variables, and additional syntactic sugaring in the form of property

[1] Given a design $\mathcal{M}$ and a temporal logic formula φ the *verification problem* asks whether $\mathcal{M}$ satisfies φ (i.e. whether φ holds on all computations of $\mathcal{M}$).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 197–212, 2013.

declarations with arguments. None of this constructs enhances the expressiveness of the language, but they do add succinctness. As for complexity, [6] show that each of these additions results in bringing the verification problem to the EXPSPACE-complete class. Yet, they conclude that the usefulness of the discussed constructs overshadows their cost, thus using them is worthwhile.

In light of this discussion we would like to distinguish between syntactic sugaring and regular expressions intersection on the one hand, and different forms of variables on the other hand. We claim that property declarations with arguments usually do not add any burden to the complexity of the verification of SVA. The EXPSPACE-hardness is obtained by deep nested declarations that expand a property of exponential size, which is not a typical usage of this feature. Similarly a sporadic usage of regular expression intersections, which is how intersections are usually used, does not significantly increase the complexity of the SVA verification problem. As in property declaration, the EXPSPACE-hardness is obtained by a long series of intersections, a rare sight of this operator.

In contrast, local variables create a complexity hurdle more easily. The upper bound of [6] is achieved by constructing an alternating Büchi automaton of size proportional to the size of the property and the size of the Cartesian product of the domains of the local variables. Seeing local variables of large domain is very common, for instance when asserting data consistency on bus protocols. Thus, a 64-bit bus results in a single variable domain of size 2^{64}. Building an alternating automaton of more than 2^{64} states as proposed in [6] may be possible, but for model checking we translate it to a non-deterministic Büchi automaton of more than $2^{2^{64}}$ states, represented by 2^{64} state variables, which is clearly infeasible.

In this paper we show that despite the theoretical lower bound shown by [6], there is a significant subset of properties with local variables for which the verification problem is in PSPACE. In particular, this subset subsumes the *simple subset* of PSL [10], which is the subset supported by most dynamic/formal verification tools for PSL/SVA.

We refer to this subset as the *practical subset*, and denote it PSL^{pract}. The precise definition of the practical subset is given in Definition 6. Intuitively, any formula with local variables limited to the monotone Boolean connectives, the temporal operators *next* , *until* , *releases* and *suffix implication* belongs to the practical subset as long as there are no assignments to local variables on either the right operand of *until* or the left operand of *releases* . The formal definition allows negation to be applied only to non-temporal expression. In order to deal with negation applied to temporal operators, we can add a pre-processing step that given a PSL formula transform it to an equivalent formula in positive normal form, by using duality rules (formally given in Definition 2).

We claim that the verification problem for the practical subset is in PSPACE, namely:

Theorem 1. *The space complexity of the verification problem of any formula φ in PSL^{pract} is polynomial in $|\varphi|$.*

The proof of Theorem 1 is done by constructing an alternating Büchi automaton $\mathcal{A}$ whose size $|\mathcal{A}|$ is polynomial in $|\varphi|$ and its language $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$. By that we reduce the verification problem of formulae to the emptiness problem of alternating Büchi automata. Our key idea is to separate the part responsible for local variables updates from the alternating automaton, and use a satellite that monitors the automaton to determine

the local variables' next state value. Then, in order to transform the alternating automaton into a non-deterministic one, we apply the Miyano-Hayashi construction [13] on an alternating automaton that is constructed for the property φ' obtained from φ by disregarding the local variables, thus avoiding the exponential penalty of the Miyano-Hayashi construction on the local-variables-part of the formula. Such a manipulation is valid conditioned in a run tree of the alternating automaton there are no two states at the same depth that disagree on the value of a local variable. We call an automaton adhering to this property *conflict-free*. Conflicts may arise because of updates after a universal branch. If the universal branch is not involved in a loop we can solve the conflict by introducing new local variables to the different branches; however, if it is involved in a loop we simply cannot. The construction for formulae of the practical subset guarantees that no local variable assignment occurs after a loop that contains a universal branch.

The idea of dealing with a subset of properties where there are no conflicts between local variables assignments appears in [12]. There as well it is observed that the complexity hurdle comes from the fact that overlapping instances, in general, may carry different values for the same local variables. And that if local variables are confined to certain positions in the property, then one can guarantee overlapping instances will agree on the value of the local variables. However, it is hard to infer from [12] what is the supported subset. Moreover, the implementation via alternating automata extended to local variables, and the separation to satellite, as well as the PSPACE claim and proof are novel to our paper.

We note that PSL^{pract} subsumes PSL^{simple}, the *simple subset* of PSL [10], which is the subset supported by most dynamic/formal verification tools for PSL/SVA. This subset conforms to the notion of monotonic advancement of time, left to right through the property. The syntactic restrictions of PSL^{pract} actually relax those of PSL^{simple}: whenever the latter requires an operand to be non-temporal, the former demands just that it does not bear assignments to local variables. We are thus confident that most commonly used properties fall into this subset.

We remark that the source of the subtle differences between the definition of local variables in PSL and SVA lies in the definition of the semantics of intersection [9] which is orthogonal to the discussion in this paper. In the absence of intersection, the semantics of SVA with local variables is exactly the same as that of PSL, except for the scope of the local variables. While PSL uses *new* and *free* to define the scope boundaries of a local variable, SVA assumes a global scope for all local variables. Thus an SVA formula of the practical subset is translatable to PSL by adding *new* for all variables at the beginning of the formula.

2 Background

We provide the syntax and semantics for the core of PSL with local variables, which we denote PSL$^{+\mathrm{V}}$. Throughout the paper we assume $\mathcal{P}$ is a non-empty set of atomic propositions and $\mathcal{V}$ is a set of local variables with domain $\mathcal{D}$. We further assume a given set $\mathcal{E}$ of (not necessarily Boolean) expressions over $\mathcal{P} \cup \mathcal{V}$, and a given set $\mathcal{B} \subseteq \mathcal{E}$ of Boolean expressions. That is, a Boolean expression may *refer* to both atomic propositions and local variables. For instance $p \wedge x$=7 is a Boolean expression stating that proposition $p \in \mathcal{P}$ holds and local variable $x \in \mathcal{V}$ has the value 7. In contrast,

assignments to local variables are not part of a Boolean expression. They are given separately as the second component of an *assignment pair* whose first component is a Boolean expression. For instance, $(b, x := 7)$ is an assignment pair that reads "the boolean expression b holds and local variable x should be assigned 7". Any expression $e \in \mathcal{E}$ can be assigned to a local variable. It is also allowed to have a sequence of local variable assignments in an assignment pair. Given a sequence of local variable $\mathrm{X} = x_1, \ldots, x_n$ and a sequence of expressions $\mathrm{E} = e_1, \ldots, e_n$ of the same length we write $\mathrm{X} := \mathrm{E}$ to abbreviate $x_1 := e_1, \ldots, x_n := e_n$. Thus, $(b, \mathrm{X} := \mathrm{E})$ is a legal assignment pair.

Formulae of PSL are defined with respect to regular expressions extended with local variables ($\mathrm{RE}^{+\mathrm{V}}$s). The atoms of an $\mathrm{RE}^{+\mathrm{V}}$ are Boolean expressions or assignment pairs. On top of these the regular operators of Concatenation, Kleene's closure and Or are applied.

We clarify that PSL/SVA with local variables are already a part of the respective standards [10,11].

2.1 Syntax of PSL$^{+\mathrm{V}}$

Definition 1 (Regular expressions extended with local variables ($\mathrm{RE}^{+\mathrm{V}}$s)). *Let $b \in \mathcal{B}$ be a Boolean expression. Let* X *be a sequence of local variables and* E *a sequence of expressions of the same length as* X*. The grammar below defines regular expressions r with local variables.*

$$r ::= b \mid (b, \mathrm{X} := \mathrm{E}) \mid \lambda \mid r \cdot r \mid r \cup r \mid r^{+} \mid (\mathit{new}(\mathrm{X})\ r) \mid (\mathit{free}(\mathrm{X})\ r)$$

Where $(b, \mathrm{X} := \mathrm{E})$ stipulates that b holds and the variables in X *are assigned with expressions in* E*, $(\mathit{new}(\mathrm{X})\ r)$ declares the local variables $x \in \mathrm{X}$ in parenthesis scope, and $(\mathit{free}(\mathrm{X})\ r)$ removes the local variables $x \in \mathrm{X}$ from parenthesis scope.*

PSL formulae are built out of $\mathrm{RE}^{+\mathrm{V}}$s. The usual negation, disjunction and conjunction can be applied to PSL formulae. The temporal operators consist of the *next* , *until* , *releases* and the *suffix implication* operator $\mapsto$ aka *triggers*. Loosely speaking $r \mapsto \varphi$ holds on a word w if every prefix of w that matches r is followed by a suffix on which φ holds.

Definition 2 (PSL$^{+\mathrm{V}}$ formulae). *Let r be an RE$^{+\mathrm{V}}$,* X *a sequence of local variables and* E *a sequence of expressions of the same length as* X*. The grammar below defines PSL$^{+\mathrm{V}}$ formulae.*

$$\varphi ::= r! \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathit{next}\ \varphi \mid \varphi\ \mathit{until}\ \varphi \mid r \mapsto \varphi \mid (\mathit{new}(\mathrm{X})\ \varphi) \mid (\mathit{free}(\mathrm{X})\ \varphi)$$

We use the following common syntactic sugaring operators: $\varphi_1 \vee \varphi_2 \stackrel{\text{def}}{=} \neg\varphi_1 \wedge \neg\varphi_2$, $\mathit{eventually}\ \varphi \stackrel{\text{def}}{=} \mathit{true}\ \mathit{until}\ \varphi$, $\mathit{always}\ \varphi \stackrel{\text{def}}{=} \neg\ \mathit{eventually}\ \neg\varphi$, $\varphi_1\ \mathit{releases}\ \varphi_2 \stackrel{\text{def}}{=} \neg(\neg\varphi_1\ \mathit{until}\ \neg\varphi_2)$, $r \circ\!\!\Rightarrow \varphi \stackrel{\text{def}}{=} \neg(r \mapsto \neg\varphi)$.

Example 1. *Let $\{start, end, data\_in, data\_out\} \subseteq \mathcal{P}$, and $x \in \mathcal{V}$. Then, the formula $(\mathit{new}(x)\ \varphi_1)$ where*

$\varphi_1 = always\ (((\neg start)^+ \cdot (start, x := data\_in)) \mapsto\ (\neg end)^+ \cdot end \wedge (data\_out = x))$

states that if the value of $data\_in$ *is* x *when transaction starts (signal* $start$ *rises) then the value of* $data\_out$ *should be* x *when the transaction ends (signal* end *rises). That is, values are transferred correctly from* $data\_in$ *to* $data\_out$.

Example 2. *Let* $\{start, end, get, put\} \subseteq \mathcal{P}$, *and* $x \in \mathcal{V}$. *Assume* put, get *and* end *are mutually exclusive (that is, if one of them holds the others do not). Let* not_pge *denote the expression* $(\neg put \wedge \neg get \wedge \neg end)$. *Then the formula* $(new(x)\ \varphi_2)$ *where*

$\varphi_2 = always\ (((start, x := 0) \cdot (not\_pge \cup (put, x{+}{+}) \cup (get, x{-}{-}))^+ \cdot end) \mapsto\ x{=}0)$

states that the number of put*s and* get*s between* $start$ *and* end *should be the same. More accurately, since the domain of variables is bounded, the number should be the same modulo the size of* $\mathcal{D}$ *assuming* $++$ *and* $--$ *increment and decrement modulo* $|\mathcal{D}|$, *respectively.*

Note that this formula is a safety *formula and it does not demand seeing the end of a transaction (signal* end *holds) once a transaction has started (signal* $start$ *holds). One thus might use instead the following* liveness *formula, which does demand that.*

$\varphi_3 = always\ ((start, x := 0) \mapsto\ ((not\_pge \cup (put, x{+}{+}) \cup (get, x{-}{-}))^+ \cdot end \cdot x{=}0))$

2.2 Semantics of PSL^{+V}

The semantics of PSL^{+V} formulae is defined with respect to a word over the alphabet $\Sigma = 2^{\mathcal{P}}$ (the set of all possible valuations of the atomic propositions) and a letter from the alphabet $\Gamma = \mathcal{D}^{\mathcal{V}}$ (the set of all possible valuations of the local variables). The semantics of RE^{+V} is defined with respect to words from the alphabet $\Lambda = \Sigma \times \Gamma \times \Gamma$. We call words over Λ *extended words*. A letter $\langle \sigma, \gamma, \gamma' \rangle$ of an extended word provides a valuation σ of the atomic propositions and two valuations γ and γ' of the local variables. The valuation γ corresponds to the value of the local variables before assignments have taken place, the *pre-value*, and the valuation γ' corresponds to the value of the local variables after they have taken place, the *post-value*.

In the sequel we use σ to denote letters from the alphabet Σ, γ to denote letters from Γ, and $\mathtt{a}$ to denote letters from Λ. We use u, v, w to denote words over Σ and $\mathtt{u}, \mathtt{v}, \mathtt{w}$ to denote words over Λ.

We use i, j and k to denote non-negative integers. We denote the i^{th} letter of v by v^{i-1} (since counting of letters starts at zero). We denote by $v^{i\cdot\cdot}$ the suffix of v starting at v^i, and by $v^{i\cdot\cdot j}$ the finite sequence of letters starting from v^i and ending at v^j. We use $|v|$ to denote the *length* of v. The empty word ϵ has length 0, a finite word $\sigma_0\sigma_1 \ldots \sigma_k$ has length $k+1$, and an infinite word v has length ∞. The notations for extended words $\mathtt{v}^i$, $\mathtt{v}^{i\cdot\cdot}$, $\mathtt{v}^{i\cdot\cdot j}$, $|\mathtt{v}|$ are defined in the same manner.

Let $\mathtt{v} = \langle \sigma_0, \gamma_0, \gamma'_0 \rangle \langle \sigma_1, \gamma_1, \gamma'_1 \rangle \cdots$ be a word over Λ. We use $\mathtt{v}|_\sigma$,$\mathtt{v}|_\gamma$,$\mathtt{v}|_{\gamma'}$ to denote the projection of $\mathtt{v}$ onto the first, second or third component, respectively, of each letter. That is, $\mathtt{v}|_\sigma = \sigma_0\sigma_1 \cdots$, $\mathtt{v}|_\gamma = \gamma_0\gamma_1 \cdots$, and $\mathtt{v}|_{\gamma'} = \gamma'_0\gamma'_1 \cdots$. We use $\mathtt{v}|_{\sigma\gamma}$ to denote the projection of $\mathtt{v}$ onto both the first and second components. That is, $\mathtt{v}|_{\sigma\gamma} = \langle \sigma_0, \gamma_0 \rangle \langle \sigma_1, \gamma_1 \rangle \cdots$. We say that an extended word $\mathtt{v}$ is *good* if for every

$i \in \{0, 1, \ldots, |\mathtt{v}| - 2\}$ we have $(\mathtt{v}|_\gamma)^{i+1} = (\mathtt{v}|_{\gamma'})^i$, i.e., the pre-value of the local variables at letter $i+1$ is the post-value at letter i.

2.2.1 Semantics of Expressions

An expression $e \in \mathcal{E}$ over $\mathcal{P} \cup \mathcal{V}$ is identified with a mapping $e : \Sigma \times \Gamma \mapsto \mathcal{D}$ where $\mathcal{D}$ is the domain of variables in $\mathcal{V}$. A Boolean expression $b \in \mathcal{B}$ is an expression whose domain is $\{\mathrm{T}, \mathrm{F}\}$, and we define *true* and *false* to be the Boolean expressions whose domains are $\{\mathrm{T}\}$ and $\{\mathrm{F}\}$, respectively.

We assume that for an atomic proposition p we have that $p(\sigma, \gamma) = \mathrm{T}$ if $p \in \sigma$ and F otherwise, and that for a local variable v we have that $v(\sigma, \gamma)$ returns the value of v in γ. We sometimes abuse notation by writing simply $p(\sigma)$ and $v(\gamma)$. We assume that operators are closed under $\mathcal{D}$ and behave in the usual manner, i.e. that for $\sigma \in \Sigma, \gamma \in \Gamma$, $e, e_1, e_2 \in \mathcal{E}$, a binary operator $\otimes$ and a unary operator $\odot$ we have $e_1(\sigma, \gamma) \otimes e_2(\sigma, \gamma) = (e_1 \otimes e_2)(\sigma, \gamma)$ and $\odot(e(\sigma, \gamma)) = (\odot e)(\sigma, \gamma)$. In particular, we assume that Boolean disjunction, conjunction and negation behave in the usual manner.

We use $:=$ for local variable *assignments*. Given a local variable x and an expression e we write $[\![x := e]\!](\sigma, \gamma)$ to denote the valuation $\hat{\gamma}$ such that $x(\hat{\gamma}) = e(\sigma, \gamma)$ and for every local variable $v \in \mathcal{V} \setminus \{x\}$ we have that $v(\hat{\gamma}) = v(\gamma)$. Sequence of assignments are evaluated left to right. Formally, given a sequence of local variable $\mathrm{X} = x_1, \ldots, x_n$ and a sequence of expressions $\mathrm{E} = e_1, \ldots, e_n$ of the same length, we write $[\![x_1 := e_1, \ldots, x_n := e_n]\!](\sigma, \gamma)$ to denote the following recursive application: $[\![x_2 := e_2, \ldots, x_n := e_n]\!](\sigma, [\![x_1 := e_1]\!](\sigma, \gamma))$. Recall that we write $\mathrm{X} := \mathrm{E}$ to abbreviate $x_1 := e_1, \ldots, x_n := e_n$. More generally, we use $\mathcal{U}$ to denote the set of all possible sequences of assignments to variables over $\mathcal{V}$. We use $\mathrm{U}, \mathrm{U}_1, \mathrm{U}_2$ to denote elements of $\mathcal{U}$ and use ε to denote the empty assignment sequence. For $\mathrm{U}_1, \mathrm{U}_2 \in \mathcal{U}$ we use $\mathrm{U}_1 \cdot \mathrm{U}_2$ to denote the application of assignments U_2 after assignments in U_1 took place.

2.2.2 Semantics of RE$^{+\mathrm{V}}$s

The semantics of RE$^{+\mathrm{V}}$ is defined with respect to a finite good word over Λ and a set of local variables $\mathrm{Z} \subseteq \mathcal{V}$, and is given in Definition 3. The role of the set Z, which is referred to as the set of *controlled variables*, is to support scoping. Any variable in Z (i.e. a variables in scope) must keep its value if not assigned and take on the assigned value otherwise, whereas any variable not in Z (i.e. a variables not in scope) is free to take on any value.

Let $\mathrm{Z} \subseteq \mathcal{V}$. We use $\gamma_1 \overset{\mathrm{Z}}{\sim} \gamma_2$ (read "γ_1 agrees with γ_2 relative to Z") to denote that for every $z \in \mathrm{Z}$ we have that $z(\gamma_1) = z(\gamma_2)$. We say that good word $\mathtt{v}$ *preserves* Z if for every $z \in \mathrm{Z}$ and for every $i < |\mathtt{v}|$ we have $z(\mathtt{v}^i|_{\gamma'}) = z(\mathtt{v}^i|_\gamma)$.

Definition 3 (Tight satisfaction). *Let b be a Boolean expression, r, r_1, r_2 RE$^{+\mathrm{V}}$s. Let* Z *be a set of local variables,* X *be a sequence of local variables and* E *a sequence of expression of same size. Let $\mathtt{v}, \mathtt{v}_1, \ldots, \mathtt{v}_k$ be good extended words. The notation $\mathtt{v} \models_{\mathrm{Z}} r$ means that $\mathtt{v}$ tightly satisfies r with respect to the controlled variables* Z*.*

- $\mathtt{v} \models_{\mathrm{Z}} b \iff |\mathtt{v}| = 1$ *and* $b(\mathtt{v}^0|_{\sigma\gamma}) = \mathrm{T}$ *and* $\mathtt{v}^0|_{\gamma'} \overset{\mathrm{Z}}{\sim} \mathtt{v}^0|_\gamma$
- $\mathtt{v} \models_{\mathrm{Z}} (b, \mathrm{X} := \mathrm{E}) \iff |\mathtt{v}| = 1$ *and* $b(\mathtt{v}^0|_{\sigma\gamma}) = \mathrm{T}$ *and* $\mathtt{v}^0|_{\gamma'} \overset{\mathrm{Z}}{\sim} [\![\mathrm{X} := \mathrm{E}]\!](\mathtt{v}^0|_{\sigma\gamma})$
- $\mathtt{v} \models_{\mathrm{Z}} \lambda \iff \mathtt{v} = \epsilon$

- $\mathtt{v} \models_{\mathrm{z}} r_1 \cdot r_2 \iff \exists \mathtt{v}_1, \mathtt{v}_2$ *such that* $\mathtt{v} = \mathtt{v}_1\mathtt{v}_2$ *and* $\mathtt{v}_1 \models_{\mathrm{z}} r_1$ *and* $\mathtt{v}_2 \models_{\mathrm{z}} r_2$
- $\mathtt{v} \models_{\mathrm{z}} r_1 \cup r_2 \iff \mathtt{v} \models_{\mathrm{z}} r_1$ *or* $\mathtt{v} \models_{\mathrm{z}} r_2$
- $\mathtt{v} \models_{\mathrm{z}} r^+ \iff \exists k \geq 1$ *and* $\mathtt{v}_1, \mathtt{v}_2, \ldots, \mathtt{v}_k$ *such that* $\mathtt{v} = \mathtt{v}_1\mathtt{v}_2 \ldots \mathtt{v}_k$ *and* $\mathtt{v}_i \models_{\mathrm{z}} r$ *for every* $1 \leq j \leq k$
- $\mathtt{v} \models_{\mathrm{z}} (\mathit{new}(\mathrm{X})\ r) \iff \mathtt{v} \models_{\mathrm{z} \cup \mathrm{x}} r$
- $\mathtt{v} \models_{\mathrm{z}} (\mathit{free}(\mathrm{X})\ r) \iff \mathtt{v} \models_{\mathrm{z} \setminus \mathrm{x}} r$

2.2.3 Semantics of Formulae

The semantics of formulae is defined with respect to a finite/infinite word over Σ, a valuation γ of the local variables and a set $\mathrm{Z} \subseteq \mathcal{V}$ of local variables. The role of Z is to support scoping, exactly as in tight satisfaction. The role of γ is to supply a current valuation of the local variables.

To connect to the semantics of $\mathrm{RE}^{+\mathrm{V}}$s which uses extended words to those of formulas which use only initial valuation of local variables, we make use of the notion of *enhancement*. An extended word $\mathtt{w}$ enhances w with respect to γ, denoted $\mathtt{w} \sqsupset \langle w, \gamma \rangle$, if $\mathtt{w}|_\sigma = w$, $\mathtt{w}$ is good, and γ is the starting pre-value, i.e. $\mathtt{w}^0|_\gamma = \gamma$. The semantic of formulas using $\mathrm{RE}^{+\mathrm{V}}$s involves quantification over the enhanced words. The quantification follows that of PSL without local variables.

Definition 4 (Satisfaction). *The notation* $\langle w, \gamma \rangle \models_{\mathrm{z}} \varphi$ *means that the word* w *satisfies* φ *with respect to controlled variables* $\mathrm{Z} \subseteq \mathcal{V}$ *and current valuation of variables* γ.

- $\langle w, \gamma \rangle \models_{\mathrm{z}} r! \iff \exists \mathtt{w} \sqsupset \langle w, \gamma \rangle$ *and* $j < |w|$ *such that* $\mathtt{w}^{0..j} \models_{\mathrm{z}} r$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} \neg\varphi \iff \langle w, \gamma \rangle \not\models_{\mathrm{z}} \varphi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} \varphi \wedge \psi \iff \langle w, \gamma \rangle \models_{\mathrm{z}} \varphi$ *and* $\langle w, \gamma \rangle \models_{\mathrm{z}} \psi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} \mathit{next}\ \varphi \iff |w| > 1$ *and* $\langle w^{1\cdot\cdot}, \gamma \rangle \models_{\mathrm{z}} \varphi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} \varphi\ \mathit{until}\ \psi \iff \exists i < |w|$, $\langle w^{i\cdot\cdot}, \gamma \rangle \models_{\mathrm{z}} \psi$ *and* $\forall j < i$, $\langle w^{j\cdot\cdot}, \gamma \rangle \models_{\mathrm{z}} \varphi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} r \Mapsto \varphi \iff \forall \mathtt{w} \sqsupset \langle w, \gamma \rangle$ *and* $j < |w|$ *such that* $\mathtt{w}^{0..j} \models_{\mathrm{z}} r$ *it holds that* $\langle w^{j+1\cdot\cdot}, \mathtt{w}^{j+1}|_\gamma \rangle \models_{\mathrm{z}} \varphi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} (\mathit{new}(\mathrm{X})\ \varphi) \iff \langle w, \gamma \rangle \models_{\mathrm{z} \cup \mathrm{x}} \varphi$
- $\langle w, \gamma \rangle \models_{\mathrm{z}} (\mathit{free}(\mathrm{X})\ \varphi) \iff \langle w, \gamma \rangle \models_{\mathrm{z} \setminus \mathrm{x}} \varphi$

Definition 5 (The Verification Problem). *Let* $\mathcal{M}$ *be a set of words over* Σ *and* γ_0 *an initial context of local variables. Let* φ *be a PSL*$^{+\mathrm{V}}$ *property and* Z *a set of local variables. We say that* $\mathcal{M}$ *satisfies* φ *with respect to* γ_0 *and* Z *if for every word* $w \in \mathcal{M}$, *we have that* $\langle w, \gamma_0 \rangle \models_{\mathrm{z}} \varphi$. *The* verification problem *is to check whether* $\mathcal{M}$ *satisfies* φ *with respect to* γ_0 *and* Z.

3 A Practical Subset of PSL$^{+\mathrm{V}}$

We define the following subset of PSL formulas with local variables, for which we will show that the complexity of the verification problem does not increase, i.e. remains in PSPACE, despite the presence of local variables.

Definition 6 (The Practical Subset, PSL^{pract}). *Let $b \in \mathcal{B}$ be a Boolean expression. Let* X *be a sequence of local variables and* E *a sequence of expressions of the same length as* X*. The grammar below defines the formulae φ that compose the practical subset, denoted PSL^{pract}:*

$$r ::= b \mid r \cdot r \mid r \cup r \mid r^+$$

$$R ::= b \mid (b, \mathrm{X} := \mathrm{E}) \mid R \cdot R \mid R \cup R \mid R^+ \mid (\mathit{new}(\mathrm{X})\ R) \mid (\mathit{free}(\mathrm{X})\ R)$$

$$\psi ::= r! \mid \neg r! \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathit{next}\ \psi \mid \psi\ \mathit{until}\ \psi \mid \psi\ \mathit{releases}\ \psi \mid r \Mapsto \psi$$

$$\varphi ::= \neg R! \mid \psi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathit{next}\ \varphi \mid \varphi\ \mathit{until}\ \psi \mid \psi\ \mathit{releases}\ \varphi \mid R \Mapsto \psi \mid (\mathit{new}(\mathrm{X})\ \varphi) \mid (\mathit{free}(\mathrm{X})\ \varphi)$$

To prove our main claim, we first need to enhance the notion of alternating automaton, with local variables. We start with a few words about (standard) alternating automata. A deterministic automaton has a single run on a give word. A *non-deterministic* automaton may have several runs on a given word. The automaton accepts a word if one of the runs is accepting (i.e. meets the acceptance condition). The dual of a non-deterministic automaton is a *universal automaton*. A universal automaton may also have several runs on a given word, but for it to accept the word all runs should be accepting. An alternating automaton combines existential and universal transitions. If Q is the set of states, a deterministic automaton maps a state and a letter to a state $q \in Q$. A non-deterministic automaton maps those to a disjunctive formula on Q e.g. $q_1 \vee q_5$. A universal automaton maps those to a conjunctive formula e.g. $q_2 \wedge q_4 \wedge q_5$. Finally, an alternating automaton maps those to a monotone formula on Q, e.g. $(q_1 \wedge (q_2 \vee q_7))$.[2] This would mean that either the runs from both q_1 and q_2 are accepting or the runs from both q_1 and q_7 are accepting.

A *local variable enhanced alternating automaton* maps a state and an extended letter to a pair whose first component is a monotone formula over Q as in a standard alternating automaton, and whose second component is sequence of local variables assignments. The intuition is that when this transition takes place, the local variables should be updated as indicated by the given assignments. For instance, if $\rho(q_1, \mathtt{a}) = \langle (q_1 \wedge (q_2 \vee q_7)), (x_1 := 2, x_2 := x_1 + 2, x_1 := 3) \rangle$ then in addition to the above 3 is assigned to x_1, and 4 to x_2.

Definition 7. A Local Variable Enhanced Alternating Automaton *over finite/infinite words defined with respect to atomic propositions $\mathcal{P}$, local variables $\mathcal{V}$ is a tuple $\mathcal{A} = \langle \mathrm{Z}, \Lambda, Q, I, \rho, F, A \rangle$, where*

- Z is the set of local variables under control,
- Λ is the extended alphabet as defined in Section 2.2,
- Q is a finite and non-empty set of states,
- $I \in \mathcal{B}^+(Q)$ describes the initial configuration of active states,
- $\rho : Q \times \Lambda \to \mathcal{B}^+(Q) \times \mathcal{U}$ gives the transition relation,

[2] Given a set Q we use $\mathcal{B}^+(Q)$ to denote the set of monotone Boolean expressions over Q.

- $F \subseteq Q$ is a subset of states for accepting *finite* words.
- $A \subseteq Q$ is a subset of states for *accepting infinite* words (the Büchi condition).

A run of a non-deterministic automaton $\mathcal{N}$ over a word is a sequence of states. A run on a finite word is accepting if it ends in a state in F. A run on an infinite word is accepting if a state in A is visited infinitely often. A word $\mathtt{w}$ is accepted provided there is an accepting run on it. A run of an alternating automaton $\mathcal{A}$ on $\mathtt{w} \in \Lambda$ is a labelled tree $\langle T, \tau \rangle$ where T is a prefix closed subset of $\mathbb{N}^*$ and τ a mapping from a node $t \in T$ to a state in Q. We use $|t|$ to denote the depth of node t in the tree T. The root of a tree is ϵ. The depth of the root $|\epsilon|$ is 0. We use $succ(t)$ to denote the successors of node t, namely the nodes $t' \in T$ such that $t' = t \cdot n$ for some $n \in \mathbb{N}$. By abuse of notations, if $succ(t) = \{t_1, \ldots, t_m\}$ we say that $\{\tau(t_1), \ldots, \tau(t_m)\}$ are the successors of state $\tau(t)$.

A labelled tree $\langle T, \tau \rangle$ is a *consistent run* of $\mathcal{A}$ on a good extended word $\mathtt{w} \in \Lambda$ and initial context $\gamma_0 \in \Gamma$ if the following three conditions hold: (a) the initial states satisfy the automaton initial condition and the initial word pre-value is γ_0, formally, $\tau(succ(\epsilon))$ satisfies I and $\mathtt{w}^0|_\gamma = \gamma_0$ and (b) the successors of a state t satisfy the transition relation with respect to $\tau(t)$ and $\mathtt{w}^{|t|}$ and (c) the post-valuation of local variables agrees with the transition's update. Formally, let $t \neq \epsilon$ be a node in T such that $\tau(t) = q$ and $\rho(q, \mathtt{w}^{|t|-1}) = \langle b, \mathrm{U} \rangle$, then for the tree to be a consistent run we should have $\tau(succ(t))$ satisfies b and $\mathtt{w}^{|t|-1}|_{\gamma'} \overset{Z}{\sim} [\![\mathrm{U}]\!](\mathtt{w}^{|t|-1}|_{\sigma\gamma})$.

The universal branches (conjunction) in the transition relation are reflected in the successor relation of $\langle T, \tau \rangle$. The existential branches (disjunction) are reflected in the fact that one may have multiple runs of $\mathcal{A}$ on $\mathtt{w}$. After a universal branch, all active successors, namely $succ(t)$, further propagate the activeness to their successors. A run $\langle T, \tau \rangle$ is *accepting* provided all paths satisfy the acceptance condition (i.e. terminate in a state in F if the path is finite, and visit infinitely often a state in A if the path is infinite). A word $\mathtt{w}$ is accepted by $\mathcal{A}$ if there exists an accepting run on it. The language of $\mathcal{A}$, denoted by $\mathcal{L}(\mathcal{A})$ is the set of words accepted by $\mathcal{A}$.

The method to prove that a property φ is valid in a model $\mathcal{M}$ by the automata-theoretic approach is to build an alternating automaton $\mathcal{A}_{\neg\varphi}$ that accepts the same language as $\neg\varphi$, namely $\mathcal{L}(\neg\varphi)$, then build the product (a.k.a. parallel composition) $\mathcal{M} \parallel \mathcal{A}_{\neg\varphi}$ and prove the language of the product automaton is empty.

Intuitively, the increase in complexity of the verification problem from PSPACE to EXPSPACE in the presence of local variables stems from the fact that the automaton needs to track all possible values a local variable may obtain. If a formula has k local variables over domain $\mathcal{D}$, the automaton should, in general, track all possible $\mathcal{D}^k$ values they may obtain. The practical subset tries to detect formulas for which the worst case will not be met. For instance, consider formula φ_1 from Example 1, intuitively the automaton should track just one value of the local variable x per each run — the value of $data\_in$ when $start$ rises.

The situation with φ_2 and φ_3 from Example 2 is more intricate. Here x may change unboundedly many times throughout the evaluation of the formula. But that in itself is not a problem – different automaton states may "record" different value of x. The problem is that transactions may overlap, i.e. $start$ may hold at cycles $k_1 < k_2$ where end does not hold in any cycle $k_1 \leq j \leq k_2$. Thus the automaton should track several possibilities for x on the same position of the same word! Think, for example, on the

word $\langle start\rangle\langle put\rangle\langle\rangle\langle put\rangle\langle put\rangle\langle start\rangle\langle put\rangle\langle\rangle\ldots$[3]. The value of x on the 8-th letter should be 4 for the first transaction but 1 for the second. However, there is an important difference between φ_2 and φ_3. To refute φ_2 it suffices to track the change on just one transaction whereas to refute φ_3 one needs to track all transaction.

Tracking different values for same position of the word means that a run tree of the automaton may have a node with two descendants that disagree on the value of the local variables. We call an automaton in which such a situation cannot occur *conflict free*. The automaton for φ_2 will be conflict-free but the one for φ_3 will not be.

Definition 8 (Conflict Free Automata). *We say that a run $\langle T, \tau\rangle$ on $\mathtt{w}$ is* conflict-free *if there exists no pair of distinct nodes $t_1, t_2 \in T$ having a common ancestor such that $\rho(\tau(t_1), \mathtt{w}^i) = \langle b_1, \mathrm{X}_1 := \mathrm{E}_1\rangle$ and $\rho(\tau(t_2), \mathtt{w}^i) = \langle b_2, \mathrm{X}_2 := \mathrm{E}_2\rangle$ where exists a local variable $z \in \mathrm{X}_1 \cap \mathrm{X}_2$. We say that $\mathcal{A}$ is* conflict-free *if every run of it is conflict-free.*[4]

Lemma 2 (Automata Construction for $\mathbf{RE^{+V}}$). *Let r be an $RE^{+\mathrm{V}}$, and $\mathrm{Z} \subseteq \mathcal{V}$. There exists a conflict-free non-deterministic finite automaton $\mathcal{N}(r)$ with $O(|r|)$ states that accept exactly the set of words $\mathtt{v}$ such that $\mathtt{v} \models_{\mathrm{Z}} r$.*

Lemma 3 (Automata Construction for the Practical Subset). *Let φ be a formula in PSL^{pract}, $\mathrm{Z} \subseteq \mathcal{V}$ a set of controlled variables, and $\gamma \in \Gamma$ an initial value for local variables. There exists a conflict-free local variable enhanced alternating word automaton $\mathcal{A}_{\gamma,\mathrm{Z}}(\neg\varphi)$ with number of states $O(|\varphi|)$ and of size $|\mathcal{A}_{\gamma,\mathrm{Z}}(\neg\varphi)| = O(|\varphi|)$ that accept exactly the set of words w such that $\langle w, \gamma\rangle \models_{\mathrm{Z}} \neg\varphi$*

Proof Sketch. In Section 4, we provide a construction for properties whose negation is in PSL^{pract}. Since the subset does not support the negation operator we need to propagate it down to $RE^{+\mathrm{V}}$s using the duality between operators, as provided at the end of Definition 2. Note that for *next* , *free*() and *new*() negation just propagates as is (that is, they are dual to themselves). Hence we end up with a property φ in the following set

$$r ::= b \mid r \cdot r \mid r \cup r \mid r^+$$

$$R ::= b \mid (b, \mathrm{X} := \mathrm{E}) \mid R \cdot R \mid R \cup R \mid R^+ \mid (\mathit{new}(\mathrm{X})\ R) \mid (\mathit{free}(\mathrm{X})\ R)$$

$$\psi ::= \neg r! \mid r! \mid \psi \vee \psi \mid \psi \wedge \psi \mid \mathit{next}\ \psi \mid \psi\ \mathit{until}\ \psi \mid \psi\ \mathit{releases}\ \psi \mid r \circledcirc \psi$$

$$\varphi ::= R! \mid \psi \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \mathit{next}\ \varphi \mid \varphi\ \mathit{releases}\ \psi \mid \psi\ \mathit{until}\ \varphi \mid R \circledcirc \psi \mid (\mathit{new}(\mathrm{X})\ \varphi) \mid (\mathit{free}(\mathrm{X})\ \varphi)$$

Following the construction in Section 4 we see that the only universal branches introduced are the ones dealing with $\wedge$, *until* and *releases* . For $\wedge$ we create distinct copies of the local variables, and for *until* and *releases* , since ψ does not contain assignments, the automaton is conflict-free. □

[3] We assume here each $\langle\cdot\rangle$ corresponds to a letter, and the content of $\langle\cdot\rangle$ specifies the propositions holding in that letter.

[4] We say that a node t' is an ancestor of node t if there exists a sequence $t'' \in \mathbb{N}^*$ such that $t = t' \cdot t''$. Note that in particular t is an ancestor of itself.

We now show that given a conflict-free alternating automaton we can extract the part dealing with local variables to a *satellite* — a machine determining the value of local variables by observing the states of the automaton and the values of atomic propositions and local variables.

Definition 9 (A Satellite). *Let $\mathcal{B}$ be an alternating automaton with state set Q, an initial value of local variables $\gamma_0 \in \Gamma$ and a set of local variables $\mathrm{Z} \subseteq \mathcal{V}$. A* satellite $\mathcal{S}$ *with respect to $\mathcal{B}$, Z and γ_0 is a set of pairs of the form (g, U) whose first element g is a Boolean expression over $\mathcal{P}$, $\mathcal{V}$ and Q; and its second element $\mathrm{U} \in \mathcal{U}$ is a local variable update.*

Intuitively, g is a condition (*guard*) upon which the assignments in U take place. Formally, let $\mathcal{B}$ be an alternating automaton over extended words, $\mathcal{S}$ a satellite as above, and $\mathtt{v}$ be an extended good word. We say that a run tree $\langle T, \tau \rangle$ of $\mathcal{B}$ on $\mathtt{v}$ is consistent with $\mathcal{S}$ if the following two conditions hold. First, the initial context agrees with γ_0, that is $\mathtt{v}^0|_\gamma = \gamma_0$. Second, for every node $t \in T$ with $|t| = i$ and $\tau(t) = q$, and every local variable $z \in \mathrm{Z}$ if there is no pair $(g, \mathrm{X} := \mathrm{E}) \in \mathcal{S}$ such that $z \in \mathrm{X}$ and $g(\mathtt{v}^i|_{\sigma\gamma}, q) = \mathrm{T}$ then $z(\mathtt{v}^i|_{\gamma'}) = z(\mathtt{v}^i|_\gamma)$, otherwise (a) there exists no other pair $(g', \mathrm{X}' := \mathrm{E}') \in \mathcal{S}$ with $z \in \mathrm{E}'$ and $g'(\mathtt{v}^i|_{\sigma\gamma}, q) = \mathrm{T}$ and (b) the word is consistent with respect to the update of z, that is, $z(\mathtt{v}^i|_{\gamma'}) = z([\![\mathrm{X} := \mathrm{E}]\!](\mathtt{v}^i|_{\sigma\gamma}))$.[5]

Lemma 4 (Satellite Extraction). *Given a conflict-free local variable-enhanced alternating word automaton $\mathcal{A}_{\gamma,\mathrm{Z}}$, there exist an alternating Büchi automaton $\mathcal{B}_\mathrm{Z}$ and a satellite $\mathcal{S}_{\gamma,\mathrm{Z}}$ with respect to $\mathcal{B}_\mathrm{Z}$ such that $|\mathcal{B}_\mathrm{Z}| + |\mathcal{S}_{\gamma,\mathrm{Z}}| = O(|\mathcal{A}_{\gamma,\mathrm{Z}}|)$ and $\mathcal{L}(\mathcal{A}_{\gamma,\mathrm{Z}}) = \mathcal{L}(\mathcal{B}_\mathrm{Z} \parallel \mathcal{S}_{\gamma,\mathrm{Z}})$.*

Proofs of Lemmas 2 and 4 are given in section 4. The proof of our main theorem follows from the above three lemmas.

Proof Sketch of Theorem 1. Given a model $\mathcal{M}$ of words (typically described as a hardware design), and a formula φ of PSL^{pract}, we check whether $\mathcal{M}$ satisfies φ as follows. Assume $\gamma \in \Gamma$ is an initial value for the local variables and Z are the variables assigned in φ. By Lemma 3 there exists a conflict-free local variable-enhanced alternating word automaton $\mathcal{A}_{\gamma,\mathrm{Z}}$ of size $O(|\varphi|)$ such that $\mathcal{L}(\mathcal{A}_{\gamma,\mathrm{Z}}) = \mathcal{L}(\neg\varphi)$.

By Lemma 4 we can construct a traditional alternating Büchi automaton $\mathcal{B}_\mathrm{Z}$ with $O(|\varphi|)$ states, and a satellite $\mathcal{S}_{\gamma,\mathrm{Z}}$ over $\mathcal{B}_\mathrm{Z}$ of size $O(|\varphi| \times |\mathrm{Z}|)$ such that $\mathcal{L}(\mathcal{S}_{\gamma,\mathrm{Z}} \parallel \mathcal{B}_\mathrm{Z}) = \mathcal{L}(\neg\varphi)$.

Since $\mathcal{B}_\mathrm{Z}$ is a traditional automaton we can apply the Miyano-Hayashi construction [13] to it to get a non-deterministic automaton $\mathcal{N}$ whose number of states is exponential in $|\varphi|$, and is representable in $O(|\varphi|)$ space, and accepts the same language as $\mathcal{B}_\mathrm{Z}$. Thus, $\mathcal{L}(\mathcal{S}_{\gamma,\emptyset} \parallel \mathcal{N}) = \mathcal{L}(\neg\varphi)$.

It follows then that $\mathcal{L}(\mathcal{M} \parallel \mathcal{S}_{\gamma,\emptyset} \parallel \mathcal{N}) = \mathcal{M} \cap \mathcal{L}(\neg\varphi)$. Thus the complexity of checking whether $\mathcal{M} \models \varphi$ reduces to the non-emptiness of non-deterministic automata

[5] For those familiar with Verilog — note that this makes the implementation of a satellite possible using a Verilog code that uses an always block with if statement for every g such that $(g, \mathrm{U}) \in \mathcal{S}$ and the body of the if statement is the sequential updates U that take place one after one at the same clock.

with number of states polynomial in $|\mathcal{M}|$ and exponential $|\varphi|$. Non-emptiness of non-deterministic automata is NLOGSPACE-Complete with respect to their size. Thus our problem can be solved in space polynomial in $|\varphi|$ and logarithmic in $|\mathcal{M}|$.

□

We can check the satisfiability of PSL^{pract} formulas, similarly — by checking the emptiness of $\mathcal{S}_{\gamma,\emptyset} \parallel \mathcal{N}$. Thus, similar arguments show that the satisfiability of PSL^{pract} can as well be checked in space polynomial in $|\varphi|$.

4 Automata Construction and Proofs

4.1 Proof of Construction for RE$^{+\mathbf{V}}$s

Proof of Lemma 2. We provide a construction of $\mathcal{N}(r)$ and then claim its correctness. For the inductive steps we assume $\mathcal{N}(r_i) = \langle Z_i, \Lambda, Q_i, I_i, \rho_i, F_i, \emptyset\rangle$ satisfy the lemma.

- $\mathcal{N}(b) = \langle \emptyset, \Lambda, \{q_0, q_{\text{ACC}}\}, q_0, \rho, q_{\text{ACC}}, \emptyset\rangle$, where
$$\rho(q_0, \mathtt{a}) = \begin{cases} \langle q_{\text{ACC}}, \varepsilon\rangle & \text{if } b(\mathtt{a}) = \mathit{true} \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$

- $\mathcal{N}(b, \mathrm{X} := \mathrm{E}) = \langle \mathrm{X}, \Lambda, \{q_0, q_{\text{ACC}}\}, q_0, \rho, q_{\text{ACC}}, \emptyset\rangle$, where
$$\rho(q_0, \mathtt{a}) = \begin{cases} \langle q_{\text{ACC}}, \mathrm{X} := \mathrm{E}\rangle & \text{if } b(\mathtt{a}) = \mathit{true} \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$

- $\mathcal{N}(\lambda) = \langle \emptyset, \Lambda, \{q_{\text{ACC}}\}, q_{\text{ACC}}, \emptyset, q_{\text{ACC}}, \emptyset\rangle$

- $\mathcal{N}(r_1 \cup r_2) = \langle \mathrm{Z}_1 \cup \mathrm{Z}_2, \Lambda, Q_1 \cup Q_2, I_1 \vee I_2, \rho_1' \cup \rho_2', F_1 \cup F_2, \emptyset\rangle$ where
$$\rho_1'(q, \mathtt{a}) = \begin{cases} \rho_1(q, \mathtt{a}) & \text{if } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$
$$\rho_2'(q, \mathtt{a}) = \begin{cases} \rho_2(q, \mathtt{a}) & \text{if } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$

- $\mathcal{N}(r_1 \cdot r_2) = \langle \mathrm{Z}_1 \cup \mathrm{Z}_2, \Lambda, Q_1 \cup Q_2, I_1, \rho_1' \cup \rho_2', F_2, \emptyset\rangle$, where
$$\rho_2'(q, \mathtt{a}) = \begin{cases} \rho_2(q, \mathtt{a}) & \text{if } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$
$$\rho_1'(q, \mathtt{a}) = \begin{cases} \langle S_1 \vee I_2, \mathrm{U}\rangle & \text{if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U}\rangle \text{ and } F_1 \cap S_1 \neq \emptyset \text{ and} \\ & \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \langle S_1, \mathrm{U}\rangle & \text{if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U}\rangle \text{ and } F_1 \cap S_1 = \emptyset \text{ and} \\ & \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \langle \mathit{false}, \varepsilon\rangle & \text{otherwise} \end{cases}$$

- $\mathcal{N}(r_1^+) = \langle \mathrm{Z}_1, \Lambda, Q_1, I_1, \rho_1', F_1, \emptyset\rangle$, where
$$\rho_1'(q, \mathtt{a}) = \begin{cases} \langle S_1 \vee I_1, \mathrm{U}\rangle & \text{if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U}\rangle \text{ and } F_1 \cap S_1 \neq \emptyset \\ \langle S_1, \mathrm{U}\rangle & \text{if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U}\rangle \text{ and } F_1 \cap S_1 = \emptyset \end{cases}$$

- $\mathcal{N}(\mathit{new}(\mathrm{X})\ r_1) = \langle \mathrm{Z}_1 \cup \mathrm{X}, \Lambda, Q_1, I_1, \rho_1, F_1, \emptyset\rangle$

- $\mathcal{N}(\mathit{free}(\mathrm{X})\ r_1) = \langle \mathrm{Z}_1 \setminus \mathrm{X}, \Lambda, Q_1, I_1, \rho_1, F_1, \emptyset\rangle$

The conflict-freeness of $\mathcal{N}$ follows trivially from the absence of universal branches in the transitions. For language acceptance, the cases λ, b, $r_1 \cdot r_2$, r_1^+ and $r_1 \cup r_2$ follow the traditional construction [3] with the desired adjustment for the set of local variables. In the cases $(\mathit{new}(\mathrm{X})\ r_1)$ and $(\mathit{free}(\mathrm{X})\ r_1)$, there are changes to Z, but no changes in the transition and the acceptance condition.

4.2 Proof of Construction for Properties of the Practical Subset

Proof of Lemma 3. For the inductive steps we assume for the operands $\varphi_1, \varphi_2, \psi_1, \psi_2$, r_1, R_1, the automata $\mathcal{A}_\gamma(\phi_i) = \langle Z_i, \Lambda, Q_i, I_i, \rho_i, F_i, A_i \rangle$ satisfy the inductive hypothesis (with $\phi \in \{r, R, \varphi, \psi\}$ and $i \in \{1, 2\}$). Let U be a sequence of assignments to local variables in Z. Let $\mathrm{X} \subseteq \mathrm{Z}$ and let X' be a set of fresh variables of same size as $|\mathrm{X}|$. We use $U[\mathrm{X} \leftarrow \mathrm{X}']$ to denote the sequence of assignments $X' := X \cdot\ U'$ where U' is obtained from U by replacing all occurrences of variables in X with the respective variable in X'. For a tuple $\langle b, U \rangle$, we use $\langle b, U \rangle[\mathrm{X} \leftarrow \mathrm{X}']$ to denote $\langle b, U[\mathrm{X} \leftarrow \mathrm{X}'] \rangle$.

- $\mathcal{A}(R_1!) = \langle \mathrm{Z}_1, \Lambda, Q_1 \cup \{q_{\text{ACC}}\}, I_1, \rho_1', F_1, \{q_{\text{ACC}}\} \rangle$, where
$$\rho_1'(q, \mathtt{a}) = \begin{cases} \langle q_{\text{ACC}}, \varepsilon \rangle & \text{if } q = q_{\text{ACC}} \\ \langle S_1, \mathrm{U} \rangle & \text{else if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U} \rangle \text{ and } F_1 \cap S_1 = \emptyset \\ \langle S_1 \vee q_{\text{ACC}}, \mathrm{U} \rangle & \text{else if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U} \rangle \text{ and } F_1 \cap S_1 \neq \emptyset \end{cases}$$

- $\mathcal{A}(\neg r_1!) = \langle \mathrm{Z}_1, \Lambda, Q_1 \cup \{q_{\text{REJ}}\}, \overline{I_1}, \rho_1', \{q_{\text{REJ}}\}, \{q_{\text{REJ}}\} \rangle$
Let b be a monotone Boolean expression. We use $\overline{b}$ to denote the Boolean expression obtained from b by replacing $\vee$ with $\wedge$ and vice versa. Let Q be a finite set $\{q_1, \ldots, q_n\}$. We use $\overline{Q}$ to denote $q_1 \wedge \cdots \wedge q_n$.
$$\rho_1'(q, \mathtt{a}) = \begin{cases} \langle q_{\text{REJ}}, \varepsilon \rangle & \text{if } q = q_{\text{REJ}} \\ \langle q_{\text{REJ}}, \varepsilon \rangle & \text{else if } \rho_1(q, \mathtt{a}) = \langle \mathit{false}, \varepsilon \rangle \\ \langle \overline{S_1}, \mathrm{U} \rangle & \text{otherwise if } \rho_1(q, \mathtt{a}) = \langle S_1, \mathrm{U} \rangle \end{cases}$$

- $\mathcal{A}(\varphi_1 \vee \varphi_2) = \langle \mathrm{Z}_1 \cup \mathrm{Z}_2, \Lambda, Q_1 \cup Q_2, I_1 \vee I_2, \rho', F_1 \cup F_2, A_1 \cup A_2 \rangle$ where
$$\rho'(q, \mathtt{a}) = \begin{cases} \rho_1(q, \mathtt{a}) & \text{if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \rho_2(q, \mathtt{a}) & \text{if } q \in Q_2 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \end{cases}$$

- $\mathcal{A}(\varphi_1 \wedge \varphi_2) = \langle \mathrm{Z}', \Lambda, Q_1 \cup Q_2, I_1 \wedge I_2, \rho', F_1 \cup F_2, A_1 \cup A_2 \rangle$
Let $\mathrm{X} = \mathrm{Z}_1 \cap \mathrm{Z}_2$, $\mathrm{Y}_1 = \mathrm{Z}_1 \setminus \mathrm{Z}_2$, $\mathrm{Y}_2 = \mathrm{Z}_2 \setminus \mathrm{Z}_1$. Let $\mathrm{X}_1, \mathrm{X}_2$ be fresh vectors of variables of same size as X.
  - $\mathrm{Z}' = \mathrm{X}_1 \cup \mathrm{Y}_1 \cup \mathrm{X}_2 \cup \mathrm{Y}_2$.
  - $\rho'(q, \mathtt{a}) = \begin{cases} \rho_1(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_1] & \text{if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \rho_2(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_2] & \text{if } q \in Q_2 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \end{cases}$

- $\mathcal{A}(\mathit{next}\ \varphi_1) = \langle \mathrm{Z}_1, \Lambda, Q_1 \cup \{q_0\}, q_0, \rho_1 \cup \rho', F_1, A_1 \rangle$, where
$\rho'(q_0, \mathtt{a}) = \langle I_1, \varepsilon \rangle$ for every $\mathtt{a} \in \Lambda$

- $\mathcal{A}(\psi_1\ \mathit{until}\ \varphi_2) = \langle \mathrm{Z}', \Lambda, Q_1 \cup Q_2 \cup \{q_0\}, I_2 \vee (I_1 \wedge q_0), \rho, F_1 \cup F_2, A_1 \cup A_2 \rangle$
Let $\mathrm{X} = \mathrm{Z}_1 \cap \mathrm{Z}_2$, $\mathrm{Y}_1 = \mathrm{Z}_1 \setminus \mathrm{Z}_2$, $\mathrm{Y}_2 = \mathrm{Z}_2 \setminus \mathrm{Z}_1$. Let $\mathrm{X}_1, \mathrm{X}_2$ be fresh vectors of variables of same size as X.

- $\mathrm{Z}' = \mathrm{X}_1 \cup \mathrm{Y}_1 \cup \mathrm{X}_2 \cup \mathrm{Y}_2$.
- $\rho(q, \mathtt{a}) = \begin{cases} \langle I_2 \vee (I_1 \wedge q_0), \varepsilon \rangle & \text{if } q \text{ is } q_0 \\ \rho_1(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_1] & \text{if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \rho_2(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_2] & \text{if } q \in Q_2 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \end{cases}$

– $\mathcal{A}(\varphi_1 \textit{ releases } \psi_2) = \langle \mathrm{Z}', \Lambda, Q_1 \cup Q_2 \cup \{q_0\}, I_2 \wedge (I_1 \vee q_0), \rho, F_1 \cup F_2, A_1 \cup A_2 \rangle$
Let $\mathrm{X} = \mathrm{Z}_1 \cap \mathrm{Z}_2$, $\mathrm{Y}_1 = \mathrm{Z}_1 \setminus \mathrm{Z}_2$, $\mathrm{Y}_2 = \mathrm{Z}_2 \setminus \mathrm{Z}_1$. Let $\mathrm{X}_1, \mathrm{X}_2$ be fresh vectors of variables of same size as X.
- $\mathrm{Z}' = \mathrm{X}_1 \cup \mathrm{Y}_1 \cup \mathrm{X}_2 \cup \mathrm{Y}_2$.
- $\rho(q, \mathtt{a}) = \begin{cases} \langle I_2 \wedge (I_1 \vee q_0), \varepsilon \rangle & \text{if } q \text{ is } q_0 \\ \rho_1(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_1] & \text{if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \rho_2(q, \mathtt{a})[\mathrm{X} \leftarrow \mathrm{X}_2] & \text{if } q \in Q_2 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \end{cases}$

– $\mathcal{A}(r_1 \circ\!\!=\!\!\diamond\ \varphi_2) = \langle \mathrm{Z}_1 \cup \mathrm{Z}_2, \Lambda, Q_1 \cup Q_2, I_1, \rho', F_2, A_2 \rangle$, where
$\rho'(q, \mathtt{a}) = \begin{cases} \langle b \vee I_2, \mathrm{U} \rangle & \text{if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \text{ and} \\ & \rho_1(q, \mathtt{a}) = \langle b, \mathrm{U} \rangle \text{ and } F_1 \cap \textit{supp}(b) \neq \emptyset \\ \rho_1(q, \mathtt{a}) & \text{otherwise if } q \in Q_1 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_2 \setminus \mathrm{Z}_1 \\ \rho_2(q, \mathtt{a}) & \text{if } q \in Q_2 \text{ and } \mathtt{a} \textit{ preserves } \mathrm{Z}_1 \setminus \mathrm{Z}_2 \end{cases}$

– $\mathcal{A}(\textit{new}(\mathrm{X})\ \varphi_1) = \langle \mathrm{Z}_1 \cup \mathrm{X}, \Lambda, Q_1, I_1, \rho_1, F_1, A_1 \rangle$

– $\mathcal{A}(\textit{free}(\mathrm{X})\ \varphi_1) = \langle \mathrm{Z}_1 \setminus \mathrm{X}, \Lambda, Q_1, I_1, \rho_1, F_1, A_1 \rangle$

We now have to show that $\mathcal{A}(\varphi)$ as constructed satisfies the premises of the lemma. It is easy to see that $\mathcal{A}(\varphi)$ is polynomial in $|\varphi|$. The proof that it accepts the same language as φ follows the one for traditional constructions [5,3]. It is left to show that it is conflict-free.

To see that it is conflict free we observe that a universal quantification is introduced in the following cases $\neg r!$, $\wedge$, *until* , *releases* . For the case of $\neg r!$ the syntax restrictions on r guarantees that r has no assignments, thus there are no updates in the generated automaton. For the case of $\wedge$ the construction introduces fresh variables for the operands, thus no conflict is met. For the constructions of the *until* and *releases* operators, the introduction of fresh variables guarantees no conflict between the updates done by the two operands. It is left to see that no conflict can arise by the several instances corresponding to the operand φ where updates of local variables are allowed. To see this note that in any run tree there will be only one layer where the successors of state q_0 (introduced in the construction for the cases of *until* and *releases*) are the initial states of $\mathcal{A}(\varphi)$. Hence, updates to the variables in φ will be done at most once in every tree. For instance, for *releases* we have that q_0 transits to $I_2 \wedge (I_1 \vee q_0)$. That is, q_0 transits to either both q_0 and I_2 or to both I_2 and I_1. So once it has chosen to transit to both I_2 and I_1 there will be no further such transitions to I_1. Since local variables updates of φ_1 will occur at the sub-tree emerging from I_1 we get that there will be only one such updates per tree.

Thus in any run of $\mathcal{A}$ if there is a local variable update to a node t at depth i then any node t' of same depth which is not a descendent of t may not have a local variable

assignment to the same variable. This is since t and t' emerge from different branches of a universal quantification, and as mentioned above, in all such cases variables of the different branches where renamed. Hence the automaton is conflict-free. □

Example 3. *Back to Example 2, formula φ_2 is in PSL^{pract} while φ_3 is not. The figure below shows the construction for both. See that in $\mathcal{A}(\neg\varphi_3)$ we have assignments taking place in a loop involving universal quantification.*

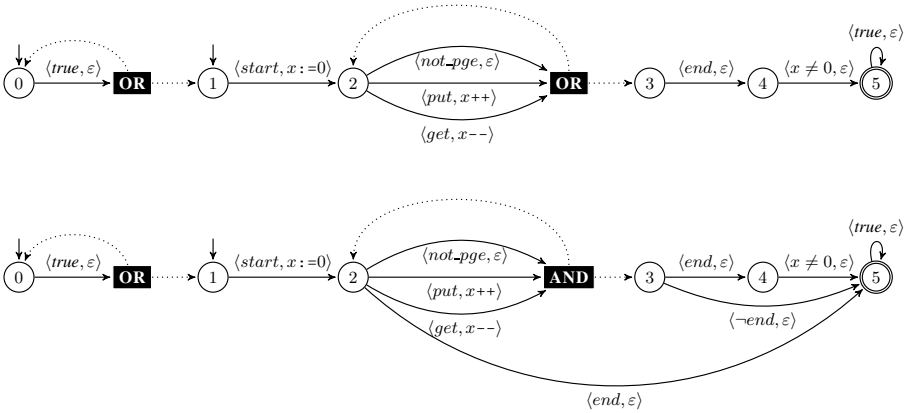


Fig. 1. Local variable enhanced alternating automata for $\neg\varphi_2$ (upper) and $\neg\varphi_3$ (lower) from Example 2

Proof of Lemma 4. Let $\mathcal{A}_{\gamma_0} = \langle \mathrm{Z}, \Lambda, Q, I, \rho, F, A\rangle$ be a conflict-free alternating Büchi automaton. We define a traditional alternating Büchi automaton $\mathcal{B}_{\mathrm{Z}}$ and a satellite $\mathcal{S}_{\gamma_0,\mathrm{Z}}$ that satisfy the theorem as follows. The automaton $\mathcal{B}_{\mathrm{Z}}$ is a traditional alternating automaton in the sense that his transitions maps to $\mathcal{B}^+(Q)$ but there are no updates of local variables associated with transitions. The automaton, does however, read the current value of local variables in its letters. It simply ignores the updates annotations of $\mathcal{A}_{\gamma_0}$. Formally, $\mathcal{B}_{\mathrm{Z}} = \langle \Sigma \times \Gamma, Q, I, \rho', F, A\rangle$ where

$$\rho'(q, \mathtt{a}) = b \quad \text{iff} \quad \rho(q, \mathtt{a}) = \langle b, \mathrm{U}\rangle \text{ for some } \mathrm{U} \in \mathcal{U}$$

The satellite $\mathcal{S}_{\gamma_0,\mathrm{Z}}$ is in-charge of making the correct updates of local variables. The definition of $\mathcal{S}_{\gamma_0,\mathrm{Z}}$ follows the transition of $\mathcal{A}_{\gamma_0}$ as follows. For every transition $\rho(q, \mathtt{a}) = \langle b, \mathrm{U}\rangle$ a pair (g, U) is added to $\mathcal{S}_{\gamma_0,\mathrm{Z}}$ where g is defined as $q \wedge b$.

To prove that $\mathcal{B}_{\mathrm{Z}}$ and $\mathcal{S}_{\gamma_0,\mathrm{Z}}$ satisfy the theorem, we show that every run of $\mathcal{A}_{\gamma_0}$ is a consistent run of $\mathcal{B}_{\mathrm{Z}}$ with respect to $\mathcal{S}_{\gamma_0,\mathrm{Z}}$. Let $\langle T, \tau\rangle$ be a run tree of $\mathcal{A}_{\gamma_0}$ on an extended word $\mathtt{w}$ with initial context γ_0. We know that the initial pre-value of $\mathtt{w}$ satisfy γ_0 and that the successors of the root satisfy the initial condition I since these requirements are the same for runs of $\mathcal{A}_{\gamma_0}$ and $\mathcal{B}_{\mathrm{Z}}$.

To see that the second condition for a consistent run is met we note that since $\mathcal{A}_{\gamma_0,\mathrm{Z}}$ is context free we are guaranteed that on every run, if there is an update to a local variable

z at node t and there is another node t' of the same depth as t then there are no updates to z from t' and any of its descendants. Thus, it cannot be that there are two pairs (g_1, U_1) and (g_2, U_2) such that both hold on a state q and they update a common variable z. Therefore the update of a local variable z will be done by the unique guard that holds at that node, if such a guard exits, and will remain the same otherwise. Therefore the second condition of a consistent run holds and $\langle T, \tau \rangle$ is a consistent run of $\mathcal{B}_{\mathrm{z}}$ with respect to $\mathcal{S}_{\gamma_0,\mathrm{z}}$.

The reasoning for the reversed direction is similar. □

References

1. Armoni, R., Fix, L., Flaisher, A., Gerth, R., Ginsburg, B., Kanza, T., Landver, A., Mador-Haim, S., Singerman, E., Tiemeyer, A., Vardi, M.Y., Zbar, Y.: The forSpec temporal logic: A new temporal property-specification language. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 296–311. Springer, Heidelberg (2002)
2. Beer, I., Ben-David, S., Eisner, C., Fisman, D., Gringauze, A., Rodeh, Y.: The temporal logic sugar. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 363–367. Springer, Heidelberg (2001)
3. Ben-David, S., Bloem, R., Fisman, D., Griesmayer, A., Pill, I., Ruah, S.: Automata construction algorithm optimized for PSL. Technical Report Delivery 3.2/4, PROSYD (July 2005)
4. Ben-David, S., Fisman, D., Ruah, S.: Embedding finite automata within regular expressions. Theor. Comput. Sci. 404(3), 202–218 (2008)
5. Bustan, D., Fisman, D., Havlicek, J.: Automata construction for PSL. Technical report, Weizmann Institute of Science (May 2005)
6. Bustan, D., Havlicek, J.: Some complexity results for systemVerilog assertions. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 205–218. Springer, Heidelberg (2006)
7. Havlicek, J., Korchemny, D., Cerny, E., Dudani, S.: Havlicek Korchemny Cerny and Dudani. Springer (2009)
8. Eisner, C., Fisman, D.: Eisner and Fisman. Springer (2006)
9. Eisner, C., Fisman, D.: Augmenting a regular expression-based temporal logic with local variables. In: Cimatti, A., Jones, R.B. (eds.) FMCAD, pp. 1–8. IEEE (2008)
10. IEEE Standard for Property Specification Language (PSL). IEEE Std 1850™-2010 (2010)
11. IEEE Standard for SystemVerilog ? Unified Hardware Design, Specification, and Verification Language. IEEE Std 1800™-2009 (2009)
12. Martensson, J.: US patent US8225249: Static formal verification of a circuit design using properties defined with local variables. Jasper Design Automation, Inc. (June 2008)
13. Miyano, S., Hayashi, T.: Alternating finite automata on omega-words. Theor. Comput. Sci. 32, 321–330 (1984)
14. Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE Computer Society (1977)

Formal Verification of Hardware Synthesis

Thomas Braibant[1] and Adam Chlipala[2]

[1] Inria Paris-Rocquencourt
[2] MIT CSAIL

Abstract. We report on the implementation of a certified compiler for a high-level hardware description language (HDL) called *Fe-Si* (FEatherweight SynthesIs). Fe-Si is a simplified version of Bluespec, an HDL based on a notion of *guarded atomic actions*. Fe-Si is defined as a dependently typed deep embedding in Coq. The target language of the compiler corresponds to a synthesisable subset of Verilog or VHDL. A key aspect of our approach is that input programs to the compiler can be defined and proved correct inside Coq. Then, we use extraction and a Verilog back-end (written in OCaml) to get a certified version of a hardware design.

Introduction

Verification of hardware designs has been thoroughly investigated, and yet, obtaining provably correct hardware of significant complexity is usually considered challenging and time-consuming. On the one hand, a common practice in hardware verification is to take a given design written in an hardware description language like Verilog or VHDL and argue about this design in a formal way using a model checker or an SMT solver. On the other hand, a completely different approach is to design hardware via a shallow embedding of circuits in a theorem prover [16,14,18,4,11]. Yet, both kinds of approach suffer from the fact that most hardware designs are expressed in low-level register transfer languages (RTL) like Verilog or VHDL, and that the level of abstraction they provide may be too low to do short and meaningful proof of high-level properties.

To raise this level of abstraction, industry moved to *high-level hardware synthesis* using higher-level languages, e.g., System-C [28], Esterel [3] or Bluespec [1], in which a source program is compiled to an RTL description. High-level synthesis has two benefits. First, it reduces the effort necessary to produce a hardware design. Second, writing or reasoning about a high-level program is simpler than reasoning about the (much more complicated) RTL description generated by a compiler. However, the downside of high-level synthesis is that there is no formal guarantee that the generated circuit description behaves exactly as prescribed by the semantics of the source program, making verification on the source program useless in the presence of compiler-introduced bugs.

In this paper, we investigate the formal verification of a lightly optimizing compiler from a Bluespec-inspired language called Fe-Si to RTL, applying (to a lesser extent) the ideas behind the CompCert project [19] to hardware synthesis.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 213–228, 2013.

Fe-Si can be seen as a stripped-down and idealized version of Bluespec: hardware designs are described in terms of *guarded atomic actions* on state elements. Guarded atomic actions have a flavour of *transactional memory*, where updates to state elements are not visible before the end of the transaction (a time-step). Our target language can be sensibly interpreted as *clocked sequential machines*: we generate an RTL description syntactically described as combinational definitions and next-state assignments. In our development, we define a (dependently typed) deep embedding of the Fe-Si programming language in Coq using *parametric higher-order abstract syntax (PHOAS)* [7], and give it a semantics using an interpreter: the semantics of a program is a Coq function that takes as inputs the current state of the state elements and produces their next state.

Fe-Si hits a sweet spot between deep and shallow embeddings: it makes it possible to use Coq as a meta programming tool to describe circuits, without the pitfalls usually associated with a deep embedding (e.g., the handling of binders). This provides an economical path toward succinct and provably correct description of, e.g., recursive circuits.

1 Overview of Fe-Si

Fe-Si is a purely functional language built around a *monad* that makes it possible to define circuits. We start with a customary example: a half adder.

```
Definition hadd (a b: Var B) : action [] (B ⊗ B) :=
  do carry ← ret (andb a b);
  do sum   ← ret (xorb a b);
  ret (carry, sum).
```

This circuit has two Boolean inputs (`Var B`) and returns a tuple of Boolean values (`B ⊗ B`). Here, we use Coq notations to implement some syntactic sugar: we borrow the `do`-notation to denote the monadic bind and use `ret` as a shorthand for return.

Up to this point, Fe-Si can be seen as an extension of the Lava [5] language, implemented in Coq rather than Haskell. Yet, using Coq as a metalanguage offers the possibility to use dependent types in our circuit descriptions. For instance, one can define an adder circuit of the following type by induction:

```
Definition adder (n: nat) (a b: Var (Int n)): action [] (Int n) := ...
```

In this definition, `n` is a formal parameter that denotes the size of the operands and the size of the result. (Note that this definition describes an infinite family of adders that can be proved correct all at once using inductive reasoning. Then, the formal parameter can be instantiated to yield certified finite-size designs.)

Stateful Programs. Fe-Si also features a small set of primitives for interacting with *memory elements* that hold mutable state. In the following snippet, we build a counter that increments its value when its input is true.

```
Definition Φ := [Reg (Int n)].
Definition count n (tick: Var B) : action Φ (Int n) :=
  do x ← !member_0;
  do _ ← if tick then {member_0 ::= x + 1} else {ret ()};
  ret x.
```

Here, Φ is an environment that defines the set of memory elements (in a broad sense) of the circuit. In the first line, we read the content of the register at position `member_0` in Φ, and bind this value to `x`. Then, we test the value of the input `tick`, and when it is true, we increment the value of the register. In any case, the output is the old value of the counter.

The above "if-then-else" construct is defined using two primitives for guarded atomic actions that are reminiscent of transactional memory monads: `assert` and `orElse`. The former aborts the current action if its argument is false. The latter takes two arguments a and b, and first executes a; if it aborts, then the effects of a are discarded and b is run. If b aborts too, the whole action a `orElse` b aborts.

Synchronous Semantics. Recall that Fe-Si programs are intended to describe hardware circuits. Hence, we must stress that they are interpreted in a synchronous setting. From a logical point of view the execution of a program (an atomic action) is clocked, and at each tick of its clock, the computation of its effects (i.e., updates to memory elements) is instantaneous: these effects are applied all at once between ticks. In particular this means that it is not possible to observe, e.g., partial updates to the memory elements, nor transient values in memory. (In Bluespec terminology, this is "reads-before-writes".)

From Programs to Circuits. At this point, the reader may wonder how it is possible to generate circuits in a palatable format out of Fe-Si programs. Indeed, using Coq as a meta-language to embed Fe-Si yields two kinds of issues. First, Coq lacks any kind of I/O; and second, a Fe-Si program may have been built using arbitrary Coq code, including, e.g., higher-order functions or fixpoints.

Note that every Coq function terminates. Therefore, a closed Fe-Si program of type `action` evaluates to a term that is syntactically built using the inductive constructors of the type `action` (i.e., all intermediate definitions in Coq have been expanded). Then we use Coq's extraction, which generates OCaml code from Coq programs. Starting from a closed Fe-Si program `foo`, we put the following definition in a Coq file:

```
Definition bar := fesic foo.
```

The extracted OCaml term that corresponds to `bar` evaluates (in OCaml) to a closed RTL circuit. Then, we can use an (unverified) back-end that pretty-prints this RTL code as regular Verilog code. (This devious use of Coq's extraction mechanism palliates the fact there is no I/O mechanism in Coq.)

2 From Fe-Si to RTL

In this section, we shall present our source (Fe-Si) and target (RTL) languages, along with their semantics. For the sake of space, we leave the full description of this compilation process out of the scope of this paper.

2.1 The Memory Model

Fe-Si programs are meant to describe sequential circuits, whose "memory footprints" must be known statically. We take a declarative approach: each state-holding element that is used in a program must be declared. We currently have three types of memory elements: inputs, registers, and register files. A register holds one value of a given type, while a register file of size n stores 2^n values of a given type. An input is a memory element that can only be read by the circuit, and whose value is driven by the external world. We show the inductive definitions of types and memory elements in Fig. 1. We have four constructors for the type `ty` of types: `Unit` (the unit type), `B` (Booleans), `Int` (integers of a given size), and `Tuple` (tuples of types). The inductive definition of memory elements (`mem`) should be self-explaining.

We endow these inductive definitions with a denotational semantics: we implement Coq functions that map such reified types to the obvious Coq types they denote.

```
Inductive ty : Type :=
| Unit : ty
| B : ty
| Int : nat → ty
| Tuple : list ty → ty.

Inductive mem : Type :=
| Input: ty →  mem
| Reg : ty → mem
| Regfile : nat → ty → mem.

Fixpoint ⟦.⟧ty : ty → Type := ...
Fixpoint ⟦.⟧mem : mem → Type := ...
Fixpoint ⟦.⟧ : list mem → Type := ...
```

Fig. 1. Types and memory elements

2.2 Fe-Si

The definition of Fe-Si programs (`action` in the following) takes the PHOAS approach [7]. That is, we define an inductive type family parameterized by an arbitrary type `V` of variables, where binders bind variables instead of arbitrary terms (as would be the case using HOAS [21]), and those variables are used explicitly via a dedicated term constructor. The definition of Fe-Si syntax is split in two syntactic classes: expressions and actions. Expressions are side-effect free and are built from variables, constants, and operations. Actions are made of control-flow structures (assertions and alternatives), binders, and memory operations.

In this work, we follow an intrinsic approach [2]: we mix the definition of the abstract syntax and the typing rules from the start. That is, the type system of the meta-language (Coq) enforces that all Fe-Si programs are well-typed by construction. Besides the obvious type-oblivious definitions (e.g., it is not possible to add a Boolean and an integer), this means that the definition of operations on state-holding elements requires some care. Here, we use dependently typed de Bruijn indices.

```
Inductive member : list mem → mem →  Type :=
| member_0 : ∀ E t, member (t:: E) t
| member_S : ∀ E t x, member E t → member (x:: E) t.
```

```
Section t.
  Variable V: ty → Type. Variable Φ: list mem.
  Inductive expr: ty → Type :=
  | Evar : ∀ t (v : V t), expr t
  | Eandb : expr B → expr B → expr B | ...                    (* operations on Booleans *)
  | Eadd : ∀ n, expr (Int n) → expr (Int n) → expr (Int n)  | ... (* operations on words *)
  | Efst : ∀ l t, expr (Tuple (t:: l)) → expr t | ...          (* operations on tuples *)

  Inductive action: ty → Type:=
  | Return: ∀ t, expr t → action t
  | Bind: ∀ t u,  action  t → (V t → action u) → action u
  (* control-flow *)
  | OrElse: ∀ t, action t → action t → action t
  | Assert: expr B → action Unit
  (* memory operations on registers *)
  | RegRead : ∀ t, member Φ (Reg t) → action t
  | RegWrite: ∀ t, member Φ (Reg t) → expr t → action Unit
  (* memory operations on register files, and inputs *)
  | ...
End t.
Definition Action Φ t := ∀ V, action V Φ t.
```

Fig. 2. The syntax of expressions and actions

Using the above definition, a term of type `member` Φ `M` denotes the fact that the memory element `M` appears at a given position in the environment of memory elements Φ. We are now ready to present the (elided) Coq definitions of the inductives for expressions and actions in Fig. 2. (For the sake of brevity, we omit the constructors for accesses to register files, in the syntax and, later, in the semantics. We refer the reader to the supplementary material [13] for more details.) Our final definition `Action` of actions is a polymorphic function from a choice of variables to an action (we refer the reader to [7] for a more in-depth explanation of this encoding strategy).

Semantics. We endow Fe-Si programs with a simple synchronous semantics: starting from an initial state, the execution of a Fe-Si program corresponds to a sequence of atomic updates to the memory elements. Each step goes as follows: reading the state, computing an update to the state, committing this update.

The reduction rules of Fe-Si programs are defined in Fig. 3. The judgement $\Gamma, \Delta \vdash a \to r$ reads "in the state Γ and with the partial update Δ, evaluating a produces the result r", where r is either `None` (meaning that the action aborted), or `Some` (`v`, Δ') (meaning that the action returned the value `v` and the partial update Δ'). Note that the PHOAS approach makes it possible to manipulate closed terms: we do not have rules for β-reduction, because it is implemented by the host language. That is, Γ only stores the mutable state, and not the variable values. There are two peculiarities here: first, following the definition of $\oplus$, if two values are written to a memory element, only the first one (in program order) is committed; second, reading a register yields the value that was held at the beginning of the time step.

$$\frac{\Gamma \vdash e \rightsquigarrow v}{\Gamma, \Delta \vdash \texttt{Return}\ e \to \texttt{Some}(v, \Delta)}$$

$$\frac{\Gamma, \Delta_1 \vdash a \to \texttt{None}}{\Gamma, \Delta_1 \vdash \texttt{Bind}\ a\ f \to \texttt{None}} \qquad \frac{\Gamma, \Delta_1 \vdash a \to \texttt{Some}\ (v, \Delta_2) \qquad \Gamma, \Delta_2 \vdash f\ v \to r}{\Gamma, \Delta_1 \vdash \texttt{Bind}\ a\ f \to r}$$

$$\frac{\Gamma \vdash e \rightsquigarrow \texttt{true}}{\Gamma, \Delta \vdash \texttt{Assert}\ e \to \texttt{Some}((), \Delta)} \qquad \frac{\Gamma \vdash e \rightsquigarrow \texttt{false}}{\Gamma, \Delta \vdash \texttt{Assert}\ e \to \texttt{None}}$$

$$\frac{\Gamma, \Delta \vdash a \to \texttt{Some}\ (v, \Delta')}{\Gamma, \Delta \vdash a\ \texttt{orElse}\ b \to \texttt{Some}\ (v, \Delta')} \qquad \frac{\Gamma, \Delta \vdash a \to \texttt{None} \qquad \Gamma, \Delta \vdash b \to r}{\Gamma, \Delta \vdash a\ \texttt{orElse}\ b \to r}$$

$$\frac{\Gamma(r) = v}{\Gamma, \Delta \vdash \texttt{RegRead}\ r \to \texttt{Some}(v, \Delta)} \qquad \frac{\Gamma \vdash e \rightsquigarrow v}{\Gamma, \Delta \vdash \texttt{RegWrite}\ r\ e \to \texttt{Some}((), \Delta \oplus (r, v))}$$

Fig. 3. Dynamic semantics of Fe-Si programs

Finally, we define a wrapper function that computes the next state of the memory elements, using the aforementioned evaluation relation (starting with an empty partial update).

```
Definition Next {t} {Φ} (st: ⟦Φ⟧) (A : Action Φ t) : option (⟦t⟧ty * ⟦Φ⟧) := ...
```

2.3 RTL

Our target language sits at the register-transfer level. At this level, a synchronous circuit can be faithfully described as a set of state-holding elements, and a next-state function, implemented using combinational logic [15]. Therefore, the definition of RTL programs (`block` in the following) is quite simple: a program is simply a well-formed sequence of bindings of expressions (combinational operations, or reads from state-holding elements), with a list of effects (i.e, writes to state-holding elements) at the end.

We show the (elided) definition of expressions and sequences of binders in Fig. 4. The definition of expressions is similar to the one we used for Fe-Si, except that we have constructors for reads from memory elements, and that we moved to "three-adress code". (That is, operands are variables, rather than arbitrary expressions.) A telescope (type `scope A`) is a well-formed sequence of binders with an element of type `A` at the end (`A` is instantiated later with a list of effects). Intuitively, this definition enforces that the first binding of a telescope can only read from memory elements; the second binding may use the first value, or read from memory elements; and so on and so forth.

A `block` is a telescope, with three elements at the end: a guard, a return value, and a (dependently typed) list of effects. The value of the guard (a Boolean) is equal to true when the return value is valid and the state updates must be committed; and false otherwise. The return value denotes the outputs of the

```
Section t.
  Variable V: ty → Type. Variable Φ: list mem.
  Inductive expr: ty → Type :=
  | Evar : ∀ t (v : V t), expr t
  (* read from memory elements *)
  | Einput : ∀ t, member Φ (Input t) → expr t
  | Eread_r : ∀ t, member Φ (Reg t) → expr t
  | Eread_rf : ∀ n t, member Φ (Regfile n t) → V (Int n) → expr t
  (* Other operations on Booleans, words, tuples, etc. *)
  | Emux : ∀ t, V B → V t → V t → expr t
  | Eandb : V B → V B → V B | ...
  | Eadd : ∀ n, V (Int n) → V (Int n) → expr (Int n) | ...
  | Efst : ∀ l t, V (Tuple (t:: l)) → expr t | ...

  Inductive scope (A : Type): Type :=
  | Send : A → scope A
  | Sbind : ∀ (t: ty), expr t → (V t → scope A) → scope A.

  Inductive write : mem → Type :=
  | WR : ∀ t, V t → V B → write (Reg t)
  | WRF : ∀ n t, V t → V (Int n) → V B →  write (Regfile n t).

  Definition effects := DList.T (option ∘ write) Φ.
  Definition block t := scope (V B * V t *  effects).
End t.
Definition Block Φ t := ∀ V, block V Φ t.
```

Fig. 4. RTL programs with three-adress code expressions

circuits. The data type `effects` encodes, for each memory element of the list Φ, either an effect (a write of the right type), or `None` (meaning that this memory element is never written to). (For the sake of brevity, we omit the particular definition of dependently typed heterogeneous lists `DList.T` that we use here.)

Semantics. We now turn to define the semantics of our RTL language. First, we endow closed expressions with a denotation function (in the same way as we did at the source level, except that it is not a recursive definition). Note that we instantiate the variable parameter of `expr` with the function $[\![.]\!]_{\texttt{ty}}$, effectively tagging variables with their denotations.

```
Variable Γ: [[Φ]].
Definition eval_expr (t : ty) (e : expr [[.]]ty t) : [[.]]ty:= ...
```

The denotation of telescopes is a simple recursive function that evaluates bindings in order and applies an arbitrary function on the final (closed) object.

```
Fixpoint eval_scope {A B} (F : A → B) (T : scope [[.]]ty A) : B := ...
```

The final piece that we need is the denotation that corresponds to the `write` type. This function takes as argument a single effect, the initial state of this memory location, and either returns a new state for this memory location, or returns `None`, meaning that location is left in its previous state.

```
Definition eval_effect (m : mem) : option (write [[.]]ty m) → [[m]]mem → option [[m]]mem := ...
```

Using all these pieces, it is quite easy to define what is the final next-state function.

```
Definition Next {t} {Φ} (Γ: [[Φ]]) (B : Block Φ t) : option ([[t]]ty * [[Φ]]) := ...
```

2.4 Compiling Fe-Si to RTL

Our syntactic translation from Fe-Si to RTL is driven by the fact that our RTL language does not allow clashing assignments: syntactically, each register and register file is updated by at most one `write` expression.

From Control Flow to Data Flow. To do so, we have to transform the control flow (the `Assert` and `OrElse`) of Fe-Si programs into data-flow. We can do that in hardware, because circuits are inherently parallel: for instance, the circuit that computes the result of the conditional expression `e ? a : b` is a circuit that computes the value of `a` and the value of `b` in parallel and then uses the value of `e` to select the proper value for the whole expression.

Administrative Normal Form. Our first compilation pass transforms Fe-Si programs into an intermediate language that implements A-normal form. That is, we assign a name to every intermediate computation. In order to do so, we also have to resolve the control flow. To be more specific, given an expression like

```
do x ← (A OrElse B); ...
```

we want to know statically to what value `x` needs to be bound and when this value is *valid*. In this particular case, we remark that if `A` yields a value v_A which is valid, then `x` needs to be bound to v_A; if `A` yields a value that is invalid, then `x` needs to be bound to the value returned by `B`. In any case, the value bound in `x` is valid whenever the value returned by `A` or the value returned by `B` is valid.

More generally, our compilation function takes as argument an arbitrary function, and returns a telescope that binds three values: (1) a *guard*, which denotes the validity of the following components of the tuple; (2) a *value*, which is bound by the telescope to denote the value that was returned by the action; (3) a list of *nested effects*, which are a lax version of the effects that exist at the `RTL` level.

The rationale behind these nested effects is to represent trees of conditional blocks, with writes to state-holding elements at the leaves. (Using this data type, several paths in such a tree may lead to a write to a given memory location; in this case, we use a notion of program order to discriminate between clashing assignments.)

Linearizing the Effects. Our second compilation pass flattens the nested effects that were introduced in the first pass. The idea of this translation is to associate two values to each register: a *data* value (the value that ought to be written) and a *write-enable* value. The data value is committed (i.e., stored) to the register if the write-enable Boolean is true. Similarly, we associate three values to each register-file: a data value, an address, and a write-enable. The data is stored to the field of the register file selected by the address if the write-enable is true.

The heart of this translation is a `merge` function that takes two writes of the same type, and returns a telescope that encapsulates a single `write`:

```
Definition merge s (a b : write s): scope (option (write s)) := ...
```

For instance, in the register case, given (v_a, e_a) (resp. (v_b, e_b)) the data value and the write-enable that correspond to `a`, the write-enable that corresponds to the merge of `a` and `b` is $e_a || e_b$, and the associated data value is $e_a \ ? \ v_a : v_b$.

Moving to RTL. The third pass of our compiler translates the previous intermediate language to RTL, which amounts to a simple transformation into three-address code. This transformation simply introduces new variables for all the intermediate expressions that appear in the computations.

2.5 Lightweight Optimizations

We will now describe two optimizations that we perform on programs expressed in the RTL language. The first one is a syntactic version of common subexpression elimination, intended to reduce the number of bindings and introduce more sharing. The second is a semantic common sub-expression elimination that aims to reduce the size of the Boolean formulas that were generated in the previous translation passes.

Syntactic Common-subexpression Elimination. We implement CSE with a simple recursive traversal of RTL programs. (Here we follow the overall approach used by Chlipala [8].) Contrary to our previous transformations that were just "pushing variables around" for each possible choice of variable representation `V`, here we need to tag variables with their symbolic values, which approximate the actual values held by variables. Then, CSE goes as follows. We fold through a telescope and maintain a mapping from symbolic values to variables. For each binder of the telescope, we compute the symbolic representation of the expression that is bound. If this symbolic value is already in the map, we avoid the creation of an extraneous binder. Otherwise, we do create a new binder, and extend our association list accordingly.

Using BDDs to Reduce Boolean Expressions. Our compilation process introduces a lot of extra Boolean variables. We use BDDs to implement semantic common-subexpression elimination. We implemented a BDD library in Coq; and we use it to annotate each Boolean expression of a program with an approximation of its runtime value, i.e., a pointer to a node in a BDD. Our use of BDDs boils down to hash-consing: it enforces that Boolean expressions that are deemed equivalent are shared.

2.6 Putting It All Together

In the end, we prove that our whole compiler is semantics preserving.

```
Variable (Φ: list mem) (t : ty).
Definition fesic  (A :  Fesi.Action Φ t) : RTL.Block Φ t :=
  let x := IR.Compile Φ  t A in let x := RTL.Compile Φ t x in
  let x := CSE.Compile Φ t x in BDD.Compile Φ t x.

Theorem fesic_correct : ∀ A (Γ : ⟦Φ⟧), Front.Next Γ A = RTL.Next Γ (fesic A).
```

A corollary of this theorem is that, given a certified Fe-Si design, our compiler produces correct by construction RTL code. Therefore, in the following, we will focus on the verification of (families of) Fe-Si designs.

3 Design and Verification of a Sorter Core

We now turn to the description of a first hardware circuit implemented and proved correct in Fe-Si. A *sorting network* [10] is a parallel sorting algorithm that sorts a sequence of values using only compare-and-swap operations, in a data-independent way. This makes it suitable for a hardware implementation.

Bitonic sorters for sequences of length 2^n can be generated using short and simple algorithmic descriptions. Yet, formally proving their correctness is a challenge that was only partially solved in two different lines of previous work. First, sorter core generators were studied from a hardware design perspective in Lava [9], but formal proof is limited to circuits with a fixed size – bounded by the performances of the automated verification tools. Second, machine-checked formal proofs of bitonic sort were performed e.g., in Agda [6], but without a connection with an actual hardware implementation. Our main contribution here is to implement such generators and to propose a formal proof of their correctness.

More precisely, we implemented a version of bitonic sort as a regular Coq program and proved that it sorts its inputs. This proof follows closely the one described by Bove and Coquand [6] – in Agda – and amounts to roughly 1000 lines of Coq, including a proof of the so-called 0-1 principle[1].

Then, we implemented a version of bitonic sort as a Fe-Si program, which mimicked the structure of the previous one. We present side-by-side the Coq implementation of `reverse` in Fig. 5. The version on the left-hand side can be seen as a specification: it takes as argument a sequence of 2^n inputs (represented as a complete binary tree of depth n) and reverses the order of this sequence. The code on the right-hand side implements part of the connection-pattern of the sorter. More precisely, it takes as input a sequence of input variables and builds a circuit that outputs this sequence in reverse order.

Next, we turn to the function that is at the heart of the bitonic sorting network. A bitonic sequence is a sequence $(x_i)_{0 \leq i < n}$ whose monotonicity changes fewer than two times, i.e.,

$$x_0 \leq \cdots \leq x_k \geq \cdots x_n, \text{with } 0 \leq k < n$$

or a circular shift of such a sequence. Given a bitonic input sequence of length 2^n, the left-hand side `min_max_swap` returns two bitonic sequences of length 2^{n-1}, such that all elements in the first sequence are smaller or equal to the elements in the second sequence. The right-hand side version of this function builds the corresponding comparator network: it takes as arguments a sequence of input variables and returns a circuit.

[1] That is, a (parametric) sorting network is valid if it sorts all sequences of 0s and 1s.

```
(* Lists of length 2^n represented as trees *)
Inductive tree (A: Type): nat → Type :=
| L : ∀ x : A, tree A 0
| N : ∀ n (l r : tree A n), tree A (S n).

Definition leaf  {A n} (t: tree A 0) : A := ...
Definition left  {A n} (t: tree A (S n)) : tree A n := ...
Definition right {A n} (t: tree A (S n)) : tree A n := ...

Fixpoint reverse {A} n (t : tree A n) :=
match t with
| L x ⇒ L x
| N n l r ⇒
  let r := (reverse n r) in
  let l := (reverse n l) in
  N n r l
end.



Variable cmp: A → A → A * A.

Fixpoint min_max_swap {A} n :
  ∀ (l r : tree A n), tree A n * tree A n :=
match n with
| 0 ⇒ fun l r ⇒
  let (x,y) := cmp (leaf l) (leaf r) in (L x, L y)
| S p ⇒ fun l r ⇒
  let (a,b) := min_max_swap p (left l) (left r) in
  let (c,d) := min_max_swap p (right l) (right r) in
  (N p a c, N p b d)
end.

...

Fixpoint sort n : tree A n → tree A n := ...
```

```
Variable A : ty.
Fixpoint domain n := match n with
| 0 ⇒ A
| S n ⇒ (domain n) ⊗ (domain n)
end.

Notation T n := tree (expr Var A) n.
Notation C n := action nil Var (domain n).

Fixpoint reverse n (t : T n) : C n  :=
match t with
| L x ⇒ ret x
| N n l r ⇒
  do r ← reverse n r;
  do l ← reverse n l;
  ret [tuple r, l]
end.

Notation mk_N x y := [tuple x,y].

Variable cmp : Var A → Var A
                 → action nil Var (A ⊗ A).
Fixpoint min_max_swap n :
  ∀ (l r : T n), C (S n) :=
match n  with
| 0 ⇒ fun l r ⇒
  cmp (leaf l) (leaf r)
| S p ⇒ fun l r ⇒
  do a,b ← min_max_swap p (left l) (left r);
  do c,d ← min_max_swap p (right l) (right r);
  ret ([tuple mk_N a c, mk_N b d])
end.

...

Fixpoint sort n : T n → C n := ...
```

Fig. 5. Comparing the specification and the Fe-Si implementation

We go on with the same ideas to finish the Fe-Si implementation of bitonic sort. The rest of the code is unsurprising, except that it requires to implement a dedicated bind operation of type

```
∀ (U: ty) n,  Var (domain n) → (T n → action [] Var U) → action [] Var U.
```

that makes it possible to recover the tree structure out of the result type of a circuit (`domain n`).

We are now ready to state (and prove) the correctness of our sorter core. We chose to fix a context where the type of data that we sort are integers of a given size, but we could generalize this proof to other data types, e.g., to sort tuples in a lexicographic order. We side-step the presentation of some of our Coq encoding, to present the final theorem in a stylized fashion.

Theorem 1. *Let I be a sequence of length 2^n of integers of size m. The circuit always produces an output sequence that is a sorted permutation of I.*

(Note that this theorem states the correctness of the Fe-Si implementation against a specification of sorted sequences that is independent of the implementation of the sorting algorithm in the left-hand side of Fig. 5; the latter only serves as a convenient intermediate step in the proof.)

Testing the Design. Finally, we indulge ourselves and test a design that was formally proven, using a stock Verilog simulator [17]. We set the word size and the number of inputs of the sorter, and we generate the corresponding Verilog code. Unsurprisingly, the sorter core sorts its input sequence in every one of our test runs.

4 Verifying a Stack Machine

The circuit that was described in the previous section is a simple combinational sorter: we could have gone one step further in this verification effort and pipelined our design by registering the output of each compare-and-swap operator. However, we chose here to describe a more interesting design: a hardware implementation of a simple stack machine for the IMP language, i.e., a tiny subset of the Java virtual machine.

Again, we proceed in two steps: first, we define a specification of the behavior of our stack machine; second, we build a Fe-Si implementation and prove that it behaves as prescribed. The instruction set of our machine is given in Fig. 6, where we let x range over identifiers (represented as natural numbers) and n, δ range over values (natural numbers). The state of the machine is composed of the code (a list of instruction), a program counter (an integer), a variable stack (a list of values), and a store (a mapping from variables to values). The semantics of the machine is given by a one-step transition relation in Fig. 6. Note that this specification uses natural numbers and lists in a pervasive manner: this cannot be faithfully encoded using finite-size machine words and register files. For simplicity reasons, we resolve this tension by adding some dynamic checks (that do not appear explicitly on Fig. 6) to the transition relation to rule out such ill-defined behaviors. (Note that this is an alternative to the use of finite-size machine words in the model; it would catch assembly programs with bugs related to overflows.)

The actual Fe-Si implementation is straightforward. The definition of the internal state is depicted below: the stack is implemented using a register file and a stack pointer; the store is a simple register file; the code is implemented as another register file that is addressed by the program counter.

```
Variable n : nat.
Notation OPCODE := (Tint 4).
Notation VAL := (Tint n).

Definition INSTR := OPCODE ⊗ VAL.
```

```
Definition Φ : state := [
Tregfile n INSTR;       (* code *)
Treg VAL;               (* program counter *)
Tregfile n VAL;         (* stack *)
Treg VAL;               (* stack pointer *)
Tregfile n VAL          (* store *)].
```

The implementation of the machine is unsurprising: we access the code memory at the address given by the program counter; we case-match over the value of the opcode and update the various elements of the machine state accordingly. For the sake of space, we only present the code for the `setvar` x instruction below.

```
Definition pop :=
do sp ← ! SP;
do x ← read STACK [: sp − 1];
do _ ← SP ::= sp − 1;
ret x.
```

```
Definition Isetvar pc i :=
do v ← pop;
do _ ← write REGS [: snd i ← v];
PC ::= pc + 1.
```

i ::= `const` n	$C \vdash pc, \sigma, s \to pc+1, n :: \sigma, s$	if $C(pc) =$ `const` n
| `var` x	$C \vdash pc, \sigma, s \to pc+1, s(x) :: \sigma, s$	if $C(pc) =$ `var` x
| `setvar` x	$C \vdash pc, v :: \sigma, s \to pc+1, \sigma, s[x \leftarrow v]$	if $C(pc) =$ `setvar` x
| `add`	$C \vdash pc, n_2 :: n_1 :: \sigma, s \to pc+1, (n_1+n_2) :: \sigma, s$	if $C(pc) =$ `add`
| `sub`	$C \vdash pc, n_2 :: n_1 :: \sigma, s \to pc+1, (n_1-n_2) :: \sigma, s$	if $C(pc) =$ `sub`
| `bfwd` δ	$C \vdash pc, \sigma, s \to pc+1+\delta, \sigma, s$	if $C(pc) =$ `bfwd` δ
| `bbwd` δ	$C \vdash pc, \sigma, s \to pc+1-\delta, \sigma, s$	if $C(pc) =$ `bbwd` δ
| `bcond` c δ	$C \vdash pc, n_2 :: n_1 :: \sigma, s \to pc+1+\delta, \sigma, s$	if $C(pc) =$ `bcond c` δ and $c\ n_1\ n_2$
	$C \vdash pc, n_2 :: n_1 :: \sigma, s \to pc+1, \sigma, s$	if $C(pc) =$ `bcond c` δ and $\neg(c\ n_1\ n_2)$
| `halt`	`no reduction`	

Fig. 6. Instruction set and transition relation of our stack machine

Correctness. We are now ready to prove that our hardware design is a sound implementation of its specification. First, we define a logical relation that relates the two encodings of machine state (in the specification and in the implementation), written $\equiv$. Then, we prove a simulation property between related states.

Theorem 2. *Let s_1 be a machine state as implemented in the specification and m_1 the machine state as implemented in the circuit, such that $s_1 \equiv m_1$. If s_1 makes a transition to s_2, then m_1 makes a transition to m_2 such that $s_2 \equiv m_2$.*

Note that we do not prove completeness here: it is indeed the case that our implementation exhibits behaviors that cannot be mapped to behaviors of the specification. In this example, the specification should be regarded as an abstraction of the actual behaviors of the implementation, which could be used to reason about the soundness of programs, either written by hand or produced by a certified compiler.

Testing the Design. Again, we compiled our Fe-Si design to an actual Verilog implementation. We load binary blobs that correspond to test programs in the code memory, and run it while monitoring the content of given memory locations. This gives rise to an highly stylized way of computing e.g., the Fibonacci sequence.

5 Comparison with Related Work

Fe-Si marries hardware design, functional programming and inductive theorem proving. We refer the reader to Sheeran [25] for a review of the use of functional programming languages in hardware design, and only discuss the most closely related work.

Lava [5] embeds parametric circuit generators in Haskell. Omitting the underlying implementation language, Lava can be described as a subset of Fe-Si, with two key differences. First, Lava features a number of layout primitives, which makes it possible to describe more precisely what should be the hardware layout, yielding more efficient FPGA implementation. We argue that these operators are irrelevant from the point of view of verification and could be added to Fe-Si if needed. Second, while Lava features "non-standard" interpretations of circuits that make it possible to prove the correctness of fixed-size tangible

representations of circuits, our embedding of Fe-Si in Coq goes further: it makes it possible to prove the correctness of parametric circuit generators.

Bluespec SystemVerilog [20] (BSV) is an industrial strength hardware description language based on non-deterministic guarded atomic actions. BSV features a module system, support for polymorphic functions and modules, support for scheduling directives, and support for static elaboration of programs (e.g., loops to express repetitive code). After static elaboration, a program is a set of rewrite rules in a Term Rewriting System that are non-deterministically executed one at a time. To implement a Bluespec program in hardware, the Bluespec compiler needs to generate a deterministic schedule where one or more rules happen each clock-cycle. Non-determinism makes it possible to use Bluespec both as an implementation language and as a specification language. Fe-Si can be described as a deterministic idealized version of Bluespec. We argue that deterministic semantics are easier to reason with, and that we can use the full power of Coq as a specification language to palliate our lack of non-determinism in the Fe-Si language. Moreover, more complex scheduling can be implemented using a few program combinators [12]: we look forward to implementing these in Fe-Si. Finally, note that using Coq as a meta language makes it possible to express (and verify) static elaboration of Fe-Si programs, similarly to what BSV provides for the core of Bluespec. We still have to determine to what extent some parts of BSV module and type systems could be either added in Fe-Si or encoded in Coq-as-a-meta-language.

The synchronous programming language Quartz [24] is part of the Averest project that aims at building tools for the development and verification of reactive systems. Quartz is a variant of Esterel that can be compiled to "guarded commands" [23]. The algorithms underlying this compiler have been verified in the HOL theorem prover, which is similar to our approach of verified high-level synthesis. However, in the Averest tool-chain, the verification (automated or interactive using HOL) takes place at the level of the guarded commands; yet, despite the naming similarity, guarded commands are closer to our RTL effects than to our guarded atomic actions. The main difference between our approaches is the place at which verification occurs: in Fe-Si, verification occurs at the level of circuit generators, written in the source language and where high-level constructs are still present; in Averest, it occurs at the intermediate representation level, in which high-level constructs have been transformed.

Richards and Lester [22] developed a shallow embedding of a subset of Bluespec in PVS. While they do not address circuit generators or the generation of RTL code, they proved the correctness of a three-input fair arbiter and a two-process implementation of Peterson's algorithm that complements our case studies (we have not attempted to translate these examples into Fe-Si).

Slind et al [26] built a compiler that creates correct-by-construction hardware implementations of arithmetic and cryptographic functions that are implemented in a synthesisable subset of HOL. Parametric circuits are not considered.

Centaur Technology and the University of Texas have developed a formal verification framework [27] that makes it possible to verify RTL and transistor

level code. They implement industrial level tools tied together in the ACL2 theorem prover and focus on hardware validation (starting from existing code). By contrast, we focus on high-level hardware synthesis, with an emphasis on the verification of parametric designs.

6 Conclusion

Our compiler is available on-line along with our examples as supplementary material [13]. The technical contributions of this paper are: a certified compiler from Fe-Si, a simple hardware description language, to RTL code; and machine checked proofs of correctness for some infinite families of hardware designs expressed in Fe-Si (parameterized by sizes), which are compiled to correct hardware designs using the above compiler.

This work is intended to be a proof of concept: much remains to be done to scale our examples to more realistic designs and to make our compiler more powerful (e.g., improving on our current optimizations). Yet, we argue that it provides an economical path to certification of parameterized hardware designs.

Acknowledgements. We thank MIT's CSG group for invaluable discussions and comments. Part of this research was done while the first author was visiting MIT from University of Grenoble. This material is based on research sponsored by DARPA under agreement number FA8750-12-2-0110.

References

1. Augustsson, L., Schwarz, J., Nikhil, R.S.: Bluespec Language definition (2001)
2. Benton, N., Hur, C.-K., Kennedy, A., McBride, C.: Strongly Typed Term Representations in Coq. J. Autom. Reasoning 49(2), 141–159 (2012)
3. Berry, G.: The foundations of Esterel. In: Proof, Language, and Interaction, Essays in Honour of Robin Milner. MIT Press (2000)
4. Beyer, S., Jacobi, C., Kröning, D., Leinenbach, D., Paul, W.J.: Putting it all together - Formal verification of the VAMP. STTT 8(4-5), 411–430 (2006)
5. Bjesse, P., Claessen, K., Sheeran, M., Singh, S.: Lava: Hardware Design in Haskell. In: Proc. ICFP, pp. 174–184. ACM Press (1998)
6. Bove, A., Coquand, T.: Formalising bitonic sort in type theory. In: Filliâtre, J.-C., Paulin-Mohring, C., Werner, B. (eds.) TYPES 2004. LNCS, vol. 3839, pp. 82–97. Springer, Heidelberg (2006)
7. Chlipala, A.: Parametric higher-order abstract syntax for mechanized semantics. In: Proc. ICFP, pp. 143–156. ACM (2008)
8. Chlipala, A.: A verified compiler for an impure functional language. In: Proc. POPL, pp. 93–106. ACM (2010)
9. Claessen, K., Sheeran, M., Singh, S.: The design and verification of a sorter core. In: Margaria, T., Melham, T.F. (eds.) CHARME 2001. LNCS, vol. 2144, pp. 355–369. Springer, Heidelberg (2001)
10. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 2nd edn. The MIT Press and McGraw-Hill Book Company (2001)

11. Coupet-Grimal, S., Jakubiec, L.: Certifying circuits in type theory. Formal Asp. Comput. 16(4), 352–373 (2004)
12. Dave, N., Arvind, Pellauer, M.: Scheduling as rule composition. In: Proc. MEMOCODE, pp. 51–60. IEEE (2007)
13. http://gallium.inria.fr/~braibant/fe-si/
14. Gordon, M.: Why Higher-Order Logic is a Good Formalism for Specifying and Verifying Hardware (1985)
15. Gordon, M.J.C.: Relating Event and Trace Semantics of Hardware Description Languages. Comput. J. 45(1), 27–36 (2002)
16. Hanna, F.K., Daeche, N., Longley, M.: Veritas$^+$: A Specification Language Based on Type Theory. In: Leeser, M., Brown, G. (eds.) Hardware Specification, Verification and Synthesis: Mathematical Aspects. LNCS, vol. 408, pp. 358–379. Springer, Heidelberg (1990)
17. http://iverilog.icarus.com/
18. Hunt Jr., W.A., Brock, B.: The Verification of a Bit-slice ALU. In: Leeser, M., Brown, G. (eds.) Hardware Specification, Verification and Synthesis: Mathematical Aspects. LNCS, vol. 408, pp. 282–306. Springer, Heidelberg (1990)
19. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
20. Nikhil, R.S., Czeck, K.R.: BSV by Example (2010)
21. Pfenning, F., Elliott, C.: Higher-Order Abstract Syntax. In: Proc. PLDI, pp. 199–208. ACM (1988)
22. Richards, D., Lester, D.R.: A monadic approach to automated reasoning for Bluespec SystemVerilog. ISSE 7(2), 85–95 (2011)
23. Schneider, K.: Embedding Imperative Synchronous Languages in Interactive Theorem Provers. In: Proc. ACSD, pp. 143–154. IEEE Computer Society (2001)
24. Schneider, K.: The Synchronous Programming Language Quartz. Technical report, University of Kaiserslautern (2010)
25. Sheeran, M.: Hardware Design and Functional Programming: a Perfect Match. J. UCS 11(7), 1135–1158 (2005)
26. Slind, K., Owens, S., Iyoda, J., Gordon, M.: Proof producing synthesis of arithmetic and cryptographic hardware. Formal Asp. Comput. 19(3), 343–362 (2007)
27. Slobodová, A., Davis, J., Swords, S., Hunt Jr., W.A.: A flexible formal verification framework for industrial scale validation. In: Proc. MEMOCODE, pp. 89–97. IEEE (2011)
28. IEEE Standard System C Language Reference Manual (2006)

CacBDD: A BDD Package with Dynamic Cache Management

Guanfeng Lv[1], Kaile Su[2,3,⋆], and Yanyan Xu[4,5]

[1] School of Comput. Sci. and Tech., Beijing University of Technology, Beijing, China
[2] IIIS, Griffith University, Brisbane, Australia
kailepku@gmail.com
[3] Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing, China
[4] School of Inf. Sci. and Tech., Beijing Forestry University, China
[5] State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing, China

Abstract. In this paper, we present CacBDD, a new efficient BDD (Binary Decision Diagrams) package. It implements a dynamic cache management algorithm, which takes account of the *hit-rate* of computed table and available memory. Experiments on the BDD benchmarks of both combinational circuits and model checking show that CacBDD is more efficient compared with the state-of-the-art BDD package CUDD.

1 Introduction

BDDs are successfully used in computer-aided verification for their efficient representation and manipulation of Boolean functions [1], and BDD packages constitute the base of some verification tools, such as NuSMV [11] and Jtlv [12]. As given in [2], classic methods of efficient implementation of BDD package include: unique table, computed table, complement edges, garbage collection and dynamic variable ordering. Besides, careful allocation of nodes can also speed up BDD packages as it reduces cache misses [4]. Even though modern BDD packages have the same cornerstone constituted by the techniques mentioned above, they may differ in a number of ways. For example, some BDD packages are pointer based, e.g. [3, 4], and some others, e.g. [5, 6, 13], use integer indices instead. As for computed table, some packages use a single computed table, and others use separate computed tables.

The unique table and computed table constitute the base data structures of modern BDD packages. The unique table is built as a hash table and contains all the BDD nodes with the hash collisions resolved by chaining. In some BDD packages, the unique table is implemented as a family of sub-tables and each of them is associated with a variable for facilitating the dynamic variable reordering [13]. The computed table (also called operation cache) is a hash-based

⋆ Corresponding author.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 229–234, 2013.

cache to record a part of the previous BDD operation results and is usually implemented without a collision chain. Complement edge is also adopted by most modern BDD packages to reduce both space and time. Besides, garbage collection and dynamic variable reordering are important for decreasing the overall size of BDDs. Nevertheless, both of them are time consuming. The overhead of garbage collection is non-negligible and dynamic variable reordering gets the lion's share of the CPU time [9].

The size of computed table has a significant impact on BDD computations in many applications, such as model checking [10]. The management of computed table is very important for a BDD package's performance, and how to find a good dynamic cache management algorithm is the first open problem given by Yang etc. [10]. Brace etc. [2] indicated that it would be easy to control the memory and run-time tradeoff by adjusting the ratio of the number of unique-table entries to the number of computed-table entries. The ratio of the above two numbers is called *hit-rate*. However, the method is still static and preliminary, and simply controlling the *hit-rate* does not work in many cases. In the well-known BDD package CUDD, a policy called "reward-based" is adopted [9]. The policy is as follows: if a large *hit-rate* of a computed table is observed, then it is worthwhile to increase the size of the computed table. Obviously, the power of the policy is limited because the algorithm considers only the *hit-rate* of a computed table, and the so-called *large* value of hit rate is not dynamically adjusted in the process of computation.

In this paper, we present CacBDD[1], a new integer-indices based BDD package. Besides careful implementations of routine techniques for efficient modern BDD packages, we adopt a new dynamic cache management method, which significantly accelerates BDD operations as indicated by the experimental results. This novel method provides a promising solution to an important open problem raised by Yang etc. [10]. Also, CacBDD has some other novel features including a new garbage collection technique.

2 Implementation

2.1 Cache Management Algorithm

CacBDD is developed in C++, which is an index-based package similar to IBM's BDD [5] and TiniBDD [6]. It supports usual operations and those useful for model checking purpose, including multiple-operand ones like AndExist. In this paper we will not discuss the details of traditional techniques used in CacBDD which can be found in [2, 4–6, 9, 14]. The main novel techniques described in this section include a dynamic cache management method and a delayed garbage collection strategy.

Computed table is used as a cache to improve BDD manipulation, and its size is limited by the available memory. It is a space and time tradeoff issue. In many applications, the *hit-rate* of computed table is a function of the instance at

[1] Available at http://kailesu.net/CacBDD

```
ite(F, G, H)
 1: ccc = ccc + 1;
 2: if (ccc >= occ) AND (cts < limitedValue) then
 3:    if (cchr >= ochr) OR (cts < nc * cchr) then
 4:       computed_table_increase_size();
 5:    end if
 6:    ochr = cchr;
 7:    occ = 2 * ccc;
 8: end if
 9: if (terminal case) then
10:    return result;
11: end if
12: if (computed-table has entry {F, G, H}) then
13:    return result;
14: else
15:    Let v be the top variable of {F, G, H};
16:    T =ite(Fv, Gv, Hv);
17:    E =ite(F v̄, G v̄, H v̄);
18:    if (T == E) then
19:       return T;
20:    end if
21:    R = find_or_add_unque_table(v, T, E);
22:    Insert_computed_table(F, G, H, R);
23:    return R;
24: end if
```

Fig. 1. The ite operation with dynamic computed table management algorithm

hand. Because the *hit-rate* of computed table is dynamic, the size of computed table should be adjusted dynamically. Therefore, if the new *hit-rate* of computed table is larger than the old one after resizing the computed table, then it should be necessary to extend the size of the computed table.

The idea above leads to our dynamic computed table management method. The algorithm for *ite* operation (the core of BDD packages) with a dynamic cache (computed table) management algorithm is give in Fig. 1.

In this algorithm, the codes from lines 1 to 8 is for the computed table management, and the remainder codes constitute the classic algorithm of *ite* operation. Number *ccc* (its initial value is 0) is the current counter of *ite* operation triggered, and number *occ* (its initial value is equal to the initial size of the computed table) serves as the threshold of *ccc* for triggering the cache management algorithm. Number *cts* is the size of computed table and *nc* is the count of the nodes. Number *cchr* is the current *hit-rate* of the computed table, and *ochr* is the last *hit-rate* of the computed table and it is initialized to 0. We also note that $limitedValue$ is the max value which can be reached by the size of the computed table, and it can be determined by the user or the BDD package according to the memory resource.

The cache management algorithm is triggered upon *occ* times running of *ite* operation. The code in line 4 is to increase the cache size, by which CacBDD doubles the size of the current cache. Accordingly, in the code of line 7, *occ* is assigned the **2** times of *ccc*. Clearly, by varying the constant **2**, we can control the frequency of triggering cache resizing. In the current version of CacBDD,

the constant is set to 2 by preliminary experiments, although it can be further tuned in the future study.

Moreover, for unary operations, if the available memory is enough, then a temporary complete operation hash is utilized by allocating a continuous array of memory of integer whose count is equal to the count of nodes. When the operation is finished, the memory block is released.

Finally, if the available memory is not enough for new added nodes, then the cache size is to shrink (by half) and the memory space is released for new added nodes. Note that the information of available physical memory can be readily obtained from modern operating systems.

2.2 Garbage Collection

Garbage collection is also a space and time tradeoff issue. In the classic BDD packages, the garbage collection is usually triggered based solely on the percentage of the dead nodes. However, the high rebirth rate indicates that garbage collection should be delayed as long as possible in some applications, such as model checking. Therefore, in CacBDD, we use a simple garbage collection triggering condition: if the free physical memory is nearly used up, then garbage collection is triggered. It is easy to see that the garbage collection triggering is almost delayed to the maximal extent.

Table 1. Comparisons with CUDD and CacBDD_Fix

Instance	CUDD		CacBDD		CacBDD_Fix		CUDD/CacBDD		CacBDD_Fix/CacBDD	
	Time	Mem	Time	Mem	Time	Mem	TR	MR	TR	MR
c2670	14.4	309.0	**4.4**	277.1	**4.4**	277.1	3.27	1.12	1.0	1.0
c3540	11.7	527.0	**5.0**	360.9	5.3	424.9	2.34	1.46	1.06	1.18
c6288-10	0.3	38.6	**0.1**	21.6	**0.1**	22.1	3.0	1.79	1.0	1.02
c6288-11	0.9	81.5	**0.5**	49.9	**0.5**	65.9	1.8	1.63	1.0	1.32
c6288-12	3.6	209.7	**2.3**	160.2	**2.3**	160.2	1.57	1.31	1.0	1.0
c6288-13	12.9	576.8	**9.7**	418.6	9.8	546.6	1.33	1.38	1.01	1.31
c6288-14	43.9	1665	**33.2**	1331.2	33.5	1331.2	1.32	1.25	1.01	1.0
c6288-15	142.2	4806.1	**103.1**	3474.4	113.4	4498.4	1.38	1.38	1.10	1.29
c6288-16	456.7	13822.8	**337.8**	10966.7	373.5	10966.7	1.35	1.26	1.11	1.0
total (1-9)	686.6	22036.5	496.1	17060.6	542.8	18293.1	1.38	1.29	1.09	1.07
abp11	**7.9**	53.5	8.5	56.8	10.9	536.8	0.93	0.94	1.28	9.45
dartes	2.0	44.4	**1.4**	81.2	1.8	25.2	1.43	0.55	1.29	0.31
dme2-16	241.7	158.2	**51.4**	343.0	121.5	335	4.70	0.46	2.36	0.98
dpd75	565.4	187.2	35.0	647.3	**34.5**	263.3	16.15	0.29	0.99	0.41
ftp3	42.0	292.4	**18.9**	570.9	34.6	318.9	2.22	0.51	1.83	0.56
furnace17	430.4	122.7	**10.3**	376.0	1274.3	128	41.79	0.33	123.72	0.34
futurebus	**255.3**	372.7	546.7	691.6	1504.2	183.6	0.47	0.54	2.75	0.27
key10	48.2	173.0	**12.9**	400.7	37.5	148.7	3.74	0.43	2.91	0.37
mmgt20	344.8	122.1	**13.9**	357.5	14.3	165.5	24.81	0.34	1.03	0.46
motors-stuck	**4.1**	64.6	7.8	58.7	10.3	30.7	0.53	1.10	1.32	0.52
over12	101.8	303.7	**24.7**	421.2	133.2	169.2	4.12	0.72	5.39	0.40
phone-async	134.8	837.9	**60.7**	1088.1	79.5	864.1	2.22	0.77	1.31	0.79
phone-sync-CW	997.8	3895.6	**825.0**	4384.9	831.3	11552.9	1.21	0.89	1.01	2.63
tcas	334.2	4878.7	182.4	3608.9	**179.7**	3224.9	1.83	1.35	0.99	0.89
tomasulo	596.1	4403.3	**188.0**	2779.3	199.5	2779.3	3.17	1.58	1.06	1.0
valves-gates	**3.0**	68.3	5.6	45.3	6.9	33.3	0.54	1.51	1.23	0.74
total (10-25)	4109.5	15978.3	1993.2	15911.4	4474	20759.4	2.06	1.00	2.24	1.30
total	4796.1	38014.8	2489.3	32972	5016.8	39052.5	1.93	1.15	2.02	1.18

3 Experimental Results

In this section, we present the experimental results, in order to demonstrate the efficiency of CacBDD and the effectiveness of the dynamic cache management algorithm in CacBDD. We compare CacBDD with CUDD (version 2.5.0), a pointer-based and publicly available BDD package. CUDD is one of the most efficient BDD packages [7, 10], and perhaps the most widely used open source package. Most importantly, CUDD is constantly updated by its author over years. To demonstrate the effectiveness of the dynamic cache management algorithm in CacBDD, we also compare CacBDD with a slightly modified version of CacBDD, called CacBDD_Fix. CacBDD_Fix is the same as CacBDD, except for that it replaces the cache management method in CacBDD with the one used by CUDD.

The benchmarks used are ISCAS85 and smv-bdd-traces98, which are representative in combinational circuits and model checking, respectively [8]. Note that the main characteristic of ISCAS85 is that all the benchmarks in it have almost the same hit-rate (nearly to 0.5) of the computed table. In contrast, the benchmarks of smv-bdd-traces98 come from different models and have different hit-rates.

The experiments are carried out on a PC workstation (Linux 64, Intel Xeon 2.80GHz CPU and 16GB RAM). We do not report the cases that are either too small (< 0.1 CPU seconds) or too large (> 16 GB of memory requirement). For fair comparison, the initial cache size is set to the same value 2^{18}. The variable order used follows the order of appearance in the file.

Table 1 reports the runtime and memory comparisons with the CUDD and CacBDD_Fix. The runtime is measured in seconds and the memory in M Bytes. TR represents the time ratio and MR the memory ratio in the table.

The experimental results show that CacBDD is more efficient than CUDD. As shown in Table 1, for all the benchmarks of ISCAS85, CacBDD is more efficient and consumes less memory than CUDD. For the benchmarks of smv-bdd-traces98, we can see that the runtime of CUDD is about as twice as that of CacBDD, while the memory usage of CUDD is almost the same for CacBDD. Especially, for *dpd75*, *furnace17*, and *mmgt20*, CacBDD achieves tens of times speedup over CUDD.

The experiments also demonstrate that the dynamic cache management in CacBDD is effective. The overall runtime performance of CacBDD is about two times better than that of CacBDD_Fix. Table 1 indicates that CacBDD is consistently better than CacBDD_Fix in both time and memory dimensions on the benchmarks of ISCAS85. For most benchmarks of smv-bdd-traces98, the time consumption of CacBDD_Fix is several times more than that of CacBDD. In particular, for *furnace17* CacBDD is 100 times faster than CacBDD_Fix.

4 Conclusion

In this paper, we presented a new BDD package (CacBDD) with a dynamic method for managing computed table. Our experimental results show that CacBDD outperforms the state-of-the-art package CUDD.

We believe that the efficiency of BDD packages can be further improved, though it seems that the classical techniques for efficient implementation of BDD packages have been mature for many years. The proposed dynamic method for computed table management can be integrated into the other existing BDD packages as future work. Also, we would like to investigate some other techniques used in BDD packages, such as hash function.

Acknowledgement. This work was Supported by ARC grants FT0991785 and DP120102489, 973 Program 2010CB328103, National Natural Science Foundation of China (61073033, 61003056 and 60903054), and Open Project SYSKF1003 of State Key Laboratory of Computer Science. We would like to thank the anonymous reviewers for their insightful comments, and Shaowei Cai and Chuan Luo for their valuable help.

References

1. Bryant, R.E.: GraphBased Algorithms for Boolean Function Manipulation. IEEE Trans. Computers, 677–691 (1986)
2. Brace, K.S., Rudell, R.L., Bryant, R.E.: Efficient Implementation of a BDD Package. In: Proc. 27th DAC 1990, pp. 40–45 (1990)
3. Somenzi, F.: CUDD: CU Decision Diagram Package Release, `http://vlsi.colorado.edu/~fabioi/`
4. Long, D.E.: The Design of a Cache-Friendly BDD Library. In: International Conference on Computer-Aided Design (ICCAD 1998), pp. 639–645 (1998)
5. Janssen, G.: Design of a Pointerless BDD Package. In: Workshop Handouts, 10th IWLS, pp. 310–315 (2001)
6. Lv, G., Su, K., Chen, Q., Chen, Y., Feng, Y.: A Succinct and Efficient Implementation of a 2^{32} BDD Package. In: The Sixth International Symposium on Theoretical Aspects of Software Engineering (TASE 2012), Beijing, China (2012)
7. Janssen, G.: A Consumer Report on BDD Packages. In: Proceedings of the 16th Symposium on Integrated Circuits and Systems Design (SBCCI 2003), Sao Paulo, Brazil (2003)
8. The BDD benchmark, `http://www.cs.cmu.edu/~bwolen/software/`
9. Somenzi, F.: Efficientmanipulation of decision diagrams. International Journal on Software Tools for Technology Transfer 3, 17–181 (2001)
10. Yang, B., Bryant, R.E., O'Hallaron, D.R., Biere, A., Coudert, O., Janssen, G., Ranjan, R.K., Somenzi, F.: A performance study of BDD-based model checking. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, pp. 255–289. Springer, Heidelberg (1998)
11. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
12. Pnueli, A., Sa'ar, Y., Zuck, L.D.: JTLV: A Framework for Developing Verification Algorithms. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 171–174. Springer, Heidelberg (2010)
13. Rudell, R.: Dynamic Variable Reordering for Ordered Binary Diagrams. In: Proc. ICCAD, pp. 139–144 (1993)
14. Armin Biere: ABCD, `http://fmv.jku.at/abcd/`

Distributed Explicit State Model Checking of Deadlock Freedom

Brad Bingham[1], Jesse Bingham[2], John Erickson[2], and Mark Greenstreet[1]

[1] University of British Columbia,
Department of Computer Science
{binghamb,mrg}@cs.ubc.ca
[2] Intel Corporation
{jesse.d.bingham,john.erickson}@intel.com

Abstract. This paper presents a practical method and associated tool for verifying deadlock freedom properties in guarded command systems. Such properties are expressed in CTL as **AGEF** q where q is a set of quiescent states. We require the user to provide transitions of the system that are "helpful" in reaching quiescent states. The distributed search constructs a path consisting of helpful transitions from each reachable state to a state that is either quiescent or is known to have a path to a quiescent state. We extended the PReach model-checker with these algorithms. Performance measurements on both academic and industrial large-scale models shows that the overhead of checking deadlock-freedom compared with state-space enumeration alone is small.

Keywords: distributed model checking, murphi, deadlock-freedom, liveness.

1 Overview

Automatic checking of liveness properties is a challenging task. Approaches to address this generally require the user to carefully specify system *fairness* assumptions that are necessary for liveness to hold. Furthermore, checking liveness is computationally expensive, being more sensitive to the state-space explosion problem than simple state-space enumeration. A broad class of liveness failures of practical importance is *deadlock*, wherein one or more transaction is blocked due to a cyclic resource dependency [1]. In such a state, there exists no path to a state where all transactions have completed; this is our motivation for characterizing deadlock-freedom by a property **AGEF** q.[1]

PReach [2,3] is a distributed explicit-state model checker for systems described in the Murφ modelling language [4]. At a high level, PReach implements the Stern-Dill algorithm [5] for message passing based, parallel state-space enumeration. This algorithm statically partitions the state-space according to a

[1] Some literature and tools identify deadlock with the much weaker property that all reachable states have at least one (possibly unique) successor. We use the stronger from, **AGEF** q throughout this paper.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 235–241, 2013.

random uniform hash function that assigns states to *owner* threads, the owner of a state being responsible for storing it once the state is *expanded*, i.e., had its successors computed. PReach is designed to be scalable, extensible and robust, and is capable of checking models with billions of states using hundreds of heterogenous machines. Several mechanisms are necessary for handling such large-scale models such as load balancing, flow control methods and state batching. The message passing layer of PReach is implemented in the distributed functional language *Erlang*, and Murφ's C++ libraries are borrowed for certain computationally demanding tasks.

A new feature added to PReach and the focus of this paper is an explicit state model checking technique to verify the CTL [6] property **AGEF** q. This property (of recent interest [7]) says "*for all reachable states, there exists a path to some q-state*". In our approach to verifying **AGEF** q, the system is modeled using guarded commands, and the user identifies a subset, $\mathcal{H}$, of these commands as *helpful*. These commands are the ones that the user expects will cause the system to make progress towards q. If s is a state and $s' \neq s$ can be reached from s by performing a helpful command, then we say that s' is a *helpful successor* of s. Thus, from any reachable state s_1,[2] we look for a *witness path*: If s_1 is a q-state, then the path is trivial; otherwise, PReach computes s_2, a helpful successor of s_1. If s_2 is a q-state then we have found a witness path for s_1; otherwise, s_3, a helpful successor of s_2 is computed. This process iterates, building a witness path $\rho = s_1, s_2, \ldots$ until a state s_i is found where either

- s_i is a q-state, or
- s_i has no helpful successor (referred to as $\mathcal{H}$-*stuck*), or
- s_i already appears in ρ.

In the first case, ρ is a witness path for s_1 and PReach continues with its standard state-space exploration algorithm. If a witness path is found for every reachable state then **AGEF** q necessarily holds. In the other two cases, PReach halts and reports the path ρ to the user. These cases do not imply $\neg$**AGEF** q. For example, if the helpful rule list is empty and there exists a reachable state that is not a q-state, then nontrivial witness paths will never be found. Likewise, PReach may choose a sequence of transitions that leads to a cycle even though a path to a q-state exists. While either error could be a false negative, as we report in Section 4, in practice such failures can show that behaviours of the model are not those intended by the designer and thereby reveal real errors.

In our experience, guarded command models have a clear partition between commands that inject new requests and those that service existing ones. Therefore, it is easy to decide suitable $\mathcal{H}$ and q. We show through experimentation that using only helpful commands to form paths is not only sufficient to verify **AGEF** q, but is *efficient* relative to the performance of state-space enumeration.

It is critical for performance to leverage the known witness paths during subsequent searches. Suppose a witness path ρ_1 has been found for state s, and s is

[2] PReach can also verify the more general property **AG** $(p \rightarrow$ **EF** $q)$, but for this paper we assume for brevity that p is *true*.

encountered on path ρ_2 while searching for witness path for s'. Clearly, the concatenation of paths ρ_1 and ρ_2 is a witness path for s'. PREACH uses a dedicated state hash table for this purpose, called **EFHT**. Each time a witness path is found for some state, it is added to **EFHT**; when we check if some state s_i is a q-state, we also check for membership in **EFHT**. Henceforth, we use $q \vee$ **EFHT** to denote states that are either q-states or members of **EFHT**.

1.1 Related Work

The idea of iteratively firing certain commands to complete in-flight transactions is very similar to the *completion functions* used by Park and Dill [8], though their goal was to verify refinement. As far as we are aware, the two tools closest to PREACH are Divine [9] and Eddy [10]. Neither of these tools are capable of checking CTL properties such as **AGEF** q, and to the best of our knowledge, neither has been applied to large scale problems as has been done with PREACH [2]. Our approach differs from the classical CTL model checking algorithm [6], which performs a pre-image fix-point computation from q to compute the set of states that satisfy **EF** q, and then checks that the reachable states are contained in this set. By using a forward search, we ensure that the space and time complexity is proportional to that of doing (explicit) state-space exploration. Doing CTL model checking only using forward searches has been investigated for symbolic model checking, e.g. the work of Iwashita et al. [11].

2 Implementation

Extending this core idea work with distributed model checking is non-trivial, and can be done in several ways. We now summarize the approaches we considered.

2.1 Local Search

This search involves no communication to other threads during a witness search. When a process in the distributed reachability algorithm encounters a new state s, the process computes a path ρ as described in Section 1. This path computation is not distributed across processors, and thus, redundant paths computions occur across different processors. While this approach scales poorly when the reachability analysis is run on a large number of machines, it provides a baseline that is free from communication overhead.

2.2 Pass-the-Path

Pass-the-path (PP) distributes the witness path searches by forwarding the current path prefix to the owner of the next state. When a state s is found in the reachability analysis, if it is already in $q \vee$ **EFHT** then a witness path is known to exist and no further work is needed. Otherwise, an enabled helpful rule is chosen; the successor state, s' is computed, and the search message $([s], s')$ is

sent to the owner of s'. Here, $[s]$ is a list of states representing the current prefix path. This process continues constructing a prefix path ρ, communicating search messages of the form (ρ, s_{cur}) to the owner of s_{cur}, until a member of $q \vee$ **EFHT** is reached, a cycle is encountered, or no helpful commands are enabled. In the first case, the owners of all states along the path are notified and they update their **EFHT**s, and otherwise a failure is reported. Notice that PP allows redundant searches and acknowledgments to occur because threads keep no record of which states have pending searches for paths to q-states.

2.3 Outstanding Search Table

These redundant searches can be avoided if threads keep track of which states have outstanding witness searches. We have implemented such an approach where each thread maintains the pending searches in local table **ST**. This approach called OST has the benefit of search messages containing only a pair of states (s, s'). If such a message arrives and there is a pending search for s' in the table, then s is added to a list of states that must be acknowledged as having a witness path once s' is acknowledged.

3 Performance

We ran PReach on a variety of combinations of Murφ models and hardware configurations, summarized in Table 1. For each, we measured the performance of

Table 1. The column "runtime" is the mean runtime of three trials, with the exception of the large cluster runs which are based on one trial (marked with †). The "overhead" columns are the additional runtime relative to that of the "no EF" mode run of the same model run on the same hardware. In PP mode, column "avg. path" is the number of helpful transitions needed to reach a state that is a member of $q \vee$ **EFHT**, averaged over the searches launched for each non-q-state. This number is *also* the average number of times each non-q-state is acknowledged for insertion to **EFHT**.

model	hardware	no EF	local	PP		OST	#states
		runtime	overhead	overhead	avg. path	overhead	
`german9`	multicore	1365.12	0.76	0.16	1.005	0.21	19844513
`flash5`	multicore	752.75	1.07	0.30	1.005	0.39	24063542
`peterson12`	multicore	4018.34	1.11	timeout	-	0.38	116039964
`mcslock6`	multicore	230.09	1.12	0.43	1.093	0.67	12838266
`german9`	small cluster	85.91	3.43	0.24	1.642	0.35	19844513
`flash5`	small cluster	57.46	1.52	0.32	1.307	0.52	24063542
`flash6`	small cluster	1854.42	1.56	0.23	1.021	0.29	609827554
`peterson12`	small cluster	254.27	9.50	timeout	-	0.50	116039964
`mcslock6`	small cluster	22.04	1.45	2.73	4.763	1.09	12838266
`intel_small`†	large cluster	1025.20	7.10	0.84	2.242	0.55	22738573
`intel_large`†	large cluster	49041.70	timeout	0.57	-	-	906695343

regular state-space enumeration (no EF), local search mode (Section 2.1), Pass-the-Path mode (Section 2.2) and Outstanding-Search-Table mode (Section 2.3). The Murφ models used are the German and Flash cache coherence protocols, the Peterson mutual exclusion algorithm, the MCS lock mutual exclusion algorithm and an Intel proprietary cache coherence protocol. We use `germanX` and `flashX` to denote these models configured with `X` caches and two data values; `peterson12` is Peterson's algorithm with 12 threads and `mcslock6` is the MCS Lock algorithm with 6 threads. All benchmarks and PReach source code is available online[3]. The compute server farms are as follows:

- multicore: 8 PReach threads on a 2 socket server machine, each processor is a Intel® Xeon® E5520 at 2.26 GHz with 4 cores.
- small cluster: 80 PReach threads on a homogenous cluster of 20 Intel Core i7-2600K at 3.40 GHz with 4 cores.
- large cluster: 100 PReach threads on a heterogenous network of contemporary Intel® Xeon® machines.

4 Summary

We have shown an efficient distributed algorithm and implementation for checking deadlock freedom properties. The simpler approach of PP does some redundant work, but performs well on models of cache coherence where paths to q-states tend to be relatively short (see Figures 1 and 2). The approach is intolerably slow for the Peterson mutual exclusion algorithm where these paths are much longer. In this case, our more involved OST algorithm has a favorable runtime and manageable memory overhead (as shown on the right of Figure 2). We find that using OST to check deadlock freedom is inexpensive – at most a 109% runtime penalty but typically much smaller.

The utility of our approach was underscored when a counterexample trace was generated on the `peterson12` benchmark. After carefully checking our definitions of q and $\mathcal{H}$, we found a critical typo in the Murφ model, which despite being

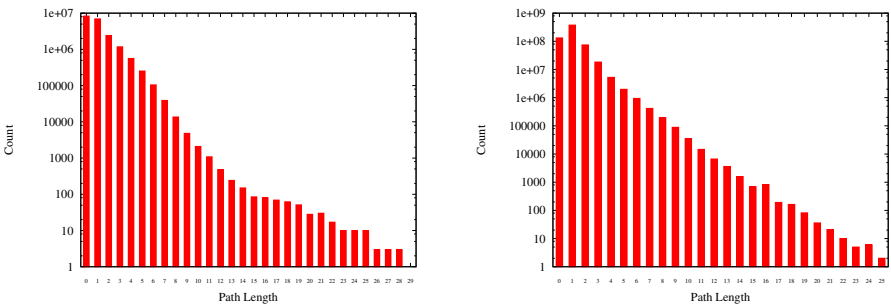


Fig. 1. Semi-log histograms of path lengths in PP mode, as defined in Table 1. The left is from `german9`/multicore, the right is from `flash6`/small cluster.

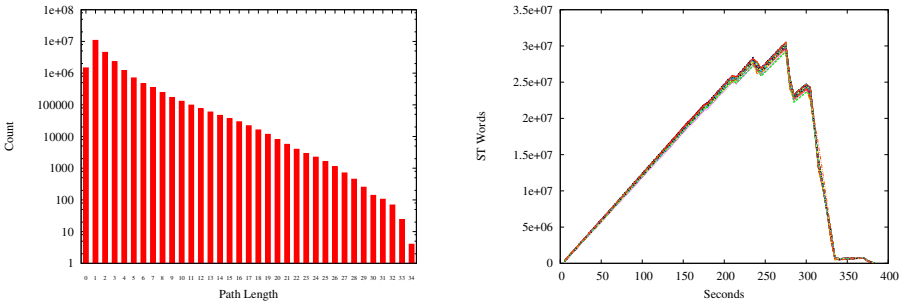


Fig. 2. Left: semi-log histogram of path lengths for `intel_small`/large cluster in PP mode. Right: memory usage of **ST** for `peterson12`/small cluster in OST mode. Here, "words" are 8 bytes; the peak memory usage over all threads is about 233 MB.

an example model in the popular Murφ distribution, has persisted for nearly 20 years. Interestingly, the bug in question was not revealed by checking the safety property mutual exclusion, or by "Murφ deadlock" (states with no successors) or *even* by checking **AGEF** q. With the bug, there exists a state where a thread attempting to enter the critical section may not do so until another thread makes the attempt first. Thus, the model does not satisfy $\mathbf{AGEF}_{\mathcal{H}}\ q$.

Checking **AGEF** q has a "buy-one, get-one free" appeal. Once model checking has been done for safety properties, a small amount of human effort is needed to identify helpful rules and write a quiesence predicate, q. While this approach cannot check for subtle liveness errors, especially ones that rely on fairness constrains, deadlocks and violations of designer intent can be found as illustrated by the Peterson example.

References

1. Holt, R.C.: Some deadlock properties of computer systems. ACM Computing Surveys 4(3), 179–196 (1972)
2. Bingham, B., Bingham, J., de Paula, F.M., Erickson, J., Singh, G., Reitblatt, M.: Industrial strength distributed explicit state model checking. In: Parallel and Distributed Model Checking (2010)
3. Bingham, B., Bingham, J., Erickson, J.: Preach online (2013), `https://bitbucket.org/binghamb/preach-brads-fork`
4. Dill, D.L.: The murphi verification system. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 390–393. Springer, Heidelberg (1996)
5. Stern, U., Dill, D.L.: Parallelizing the murphi verifier. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 256–278. Springer, Heidelberg (1997)
6. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
7. Hassan, Z., Bradley, A.R., Somenzi, F.: Incremental, inductive CTL model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 532–547. Springer, Heidelberg (2012)

8. Park, S., Dill, D.L.: Protocol verification by aggregation of distributed transactions. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 300–310. Springer, Heidelberg (1996)
9. Barnat, J., Brim, L., Češka, M., Lamr, T.: CUDA accelerated LTL Model Checking. In: ICPADS 2009. IEEE (2009)
10. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. Int'l. J. Softw. Tools Technol. Transf. 11(1), 13–25 (2009)
11. Iwashita, H., Nakata, T., Hirose, F.: CTL model checking based on forward state traversal. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 82–87. Springer, Heidelberg (1996)

Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems*

Hui Kong[1,2,5,6], Fei He[2,5,6], Xiaoyu Song[3], William N.N. Hung[4], and Ming Gu[2,5,6]

[1] Dept. of Computer Science&Technology, Tsinghua University, Beijing, China
[2] School of Software, Tsinghua University, Beijing, China
[3] Dept. of ECE, Portland State University, Oregon, USA
[4] Synopsys Inc, Mountain View, California, USA
[5] Tsinghua National Laboratory for Information Science and Technology
[6] Key Laboratory for Information System Security, MOE, China

Abstract. A barrier certificate is an inductive invariant function which can be used for the safety verification of a hybrid system. Safety verification based on barrier certificate has the benefit of avoiding explicit computation of the exact reachable set which is usually intractable for nonlinear hybrid systems. In this paper, we propose a new barrier certificate condition, called *Exponential Condition*, for the safety verification of semi-algebraic hybrid systems. The most important benefit of *Exponential Condition* is that it has a lower conservativeness than the existing convex conditions and meanwhile it possesses the convexity. On the one hand, a less conservative barrier certificate forms a tighter over-approximation for the reachable set and hence is able to verify critical safety properties. On the other hand, the convexity guarantees its solvability by semidefinite programming method. Some examples are presented to illustrate the effectiveness and practicality of our method.

Keywords: inductive invariant, barrier certificate, safety verification, hybrid system, nonlinear system, sum of squares.

1 Introduction

Hybrid systems [1], [2] are models for those systems with interacting discrete and continuous dynamics. Embedded systems are often modeled as hybrid systems due to their involvement of both digital control software and analog plants. In recent years, as embedded systems are becoming ubiquitous, more and more researchers are devoted to the theory of hybrid systems. Reachability problems or

* This work was supported by the Chinese National 973 Plan under grant No. 2010CB328003, the NSF of China under grants No. 61272001, 60903030, 91218302, the Chinese National Key Technology R&D Program under grant No. SQ2012BAJY4052, and the Tsinghua University Initiative Scientific Research Program.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 242–257, 2013.

safety verification problems are among the most challenging problems in verifying hybrid systems. The aim of safety verification is to decide that starting from an initial set, whether a continuous system or hybrid system can reach an unsafe set. For this purpose, many methods have been proposed for various hybrid systems with different features.

Deductive methods based on inductive invariant play an important role in the verification of hybrid systems. An inductive invariant of a hybrid system is an invariant φ that holds at the initial states of the system, and is preserved by all discrete and continuous transitions. A safety property is an invariant ψ (usually not inductive) that holds in all reachable states of the system. The standard technique for proving a given property ψ is to generate an inductive invariant φ that implies ψ. Therefore, the problem of safety verification is converted to the problem of inductive invariant generation and hence avoid the reachability computation of the hybrid system. The key points in generating inductive invariant for hybrid systems is how to define an inductive condition that is the least conservative and how to efficiently compute the inductive invariant that satisfies the inductive condition. Usually, these two aspects contradict with each other, that is, an inductive condition with sufficiently low conservativeness often encounters the computability or complexity problem. For different class of hybrid systems, various inductive invariants and computational methods have been proposed.

Some methods were primarily proposed for constructing inductive invariant for linear hybrid systems [3], [4]. In recent years, however, researchers concentrate more and more on nonlinear hybrid systems, especially on algebraic or semi-algebraic hybrid systems (i.e. those systems whose vector fields are polynomials and whose set descriptions are polynomial equalities or inequalities), as they have a higher universality. In [5], [6], Sankaranarayanan et al. presented a computational method based on the theory of ideal over polynomial ring and quantifier elimination for automatically generating algebraic invariants for algebraic hybrid systems. Similarly, Tiwari et al. proposed in [7] a technique based on the theory of ideal over polynomial ring to generate the inductive invariant for nonlinear polynomial systems. In [8], [9], S. Prajna et al. proposed a new inductive invariant called *Barrier Certificate* for verifying the safety of semialgebraic hybrid systems and the computational method they applied is the technique of sum-of-squares decomposition of semidefinite polynomials. In [10], C. Sloth et al. proposed a new *Barrier Certificate* for a special class of hybrid systems which can be modeled as an interconnection of subsystems. In [11], A. Platzer et al. proposed the concept of *Differential Invariant* which is a boolean combination of multiple polynomial inequalities for verifying semialgebraic hybrid systems. In [12], S. Gulwani et al. proposed an inductive invariant similar to *Differential Invariant* except that they defined a different inductive condition and they used SMT solver to solve the inductive invariant. In [13], A. Taly et al. discussed the soundness and completeness of several existing invariant condition and presented several simpler and practical invariant condition that are sound and relatively complete for different classes of inductive invariants. In [14], A. Taly et al.

proposed to use inductive controlled invariant to synthesize multi-modal continuous dynamical systems satisfying a specified safety property.

In this paper, we propose a new barrier certificate (called *Exponential Condition*) for the safety verification of semialgebraic hybrid systems. A barrier certificate is a special class of inductive invariant for the safety verification of hybrid systems: a function $\varphi(x)$ which maps all the states in the reachable set to non-positive reals and all the states in the unsafe set to positive reals. Given a dynamical system S with dynamics $\dot{x} = f(x)$ with initial set $Init$, to prove a safety property P (we use X_u to denote the unsafe set) is satisfied by S, the basic idea of *Exponential Condition* is to identify a function $\varphi(x)$ such that 1) $\varphi(x) \leq 0$ for any point $x \in Init$, 2) $\varphi(x) > 0$ for any point $x \in X_u$, and 3) $\mathcal{L}_f\varphi(x) \leq \lambda\varphi(x)$, where $\mathcal{L}_f\varphi(x) = \frac{\partial\varphi}{\partial x}f(x) = \sum_{i=1}^{n}\frac{\partial\varphi}{\partial x_i}f_i(x)$ is the Lie derivative of φ with respect to the vector field f and λ is any negative constant real value. The first condition and the third condition together guarantee that $\varphi(x) \leq 0$ for any point x in the reachable set R, which implies that $R \cap X_u = \emptyset$. Therefore, we can assert that the safety property P is satisfied by the system M as long as we can find a function $\varphi(x)$ satisfying the above condition. The above condition can be extended to semialgebraic hybrid systems naturally. The idea is to identify a set of functions $\{\varphi_i(x)\}$, one for each mode of the hybrid system, which not only satisfy the above condition but also satisfy an additional sign-preserving constraint for each discrete transition.

The most important benefit of *Exponential Condition* is that it is less conservative than *Convex Condition* [8] and *Differential Invariant* [11], where the Lie derivative of $\varphi(x)$ is required to satisfy that $\mathcal{L}_f\varphi(x) \leq 0$ (a stronger condition than $\mathcal{L}_f\varphi(x) \leq \lambda\varphi(x)$), and meanwhile, it possesses the property of convexity as well. On the one hand, a less conservative inductive invariant forms a tighter over-approximation for the reachable set and hence is able to verify critical safety properties (i.e., the unsafe region is very close to reachable region). On the other hand, a convex inductive invariant condition can be solved efficiently by semidefinite programming method, which is widely used for computing Lyapunov functions in the stability analysis of nonlinear systems. In fact, there exist some other less conservative inductive invariants than *Exponential Condition*, such as [8], [12], [13], however, these inductive conditions are not convex and thus cannot be solved by semidefinite programming method. Instead, they are usually solved by quantifier elimination and SMT solver, which usually has a much higher computational complexity than semidefinite programming method.

Given a semialgebraic hybrid system, we choose a set of polynomials of bounded degree with unknown coefficients as the candidate inductive invariant, and then we obtain a set of positive semidefinite polynomials (i.e. $P(x) \geq 0$) according to *Exponential Condition*. Therefore, the generation of barrier certificate based on *Exponential Condition* can be transformed to the problem of sum-of-squares programming of positive semidefinite polynomials [15]. Based on our theory, we develop an algorithm for generating the inductive invariant satisfying *Exponential Condition*. Experiments on both nonlinear systems and hybrid systems show the effectiveness and practicality of our method.

The remainder of this paper is organized as follows. Section 2 introduces the preliminaries of our method. Section 3 presents the barrier certificate conditions for continuous systems and hybrid systems. Section 4 introduces the computational method we use to construct barrier certificates according to the barrier certificate conditions. Section 5 gives some examples to demonstrate the application of our method to the safety verification of continuous and hybrid systems. Finally, we conclude our work in Section 6.

2 Preliminaries

In this paper, we adopt the model proposed in [16] as our modeling framework. Many other models for hybrid system can be found in [17], [2].

A continuous system is specified by a differential equation

$$\dot{x} = f(x) \tag{1}$$

where $x \in \mathbb{R}^n$ and f is a Lipschitz continuous vector function from $\mathbb{R}^n$ to $\mathbb{R}^n$. Note that the Lipschitz continuity guarantees the existence and uniqueness of the solution $x(t)$ to the system (1). A hybrid system can then be defined as:

Definition 1. ***(Hybrid System)*** *A hybrid system is a tuple* $\mathcal{H} = \langle L, X, E, R, G, I, F \rangle$*, where*

- L *is a finite set of locations (or modes);*
- $X \subseteq \mathbb{R}^n$ *is the continuous state space. The hybrid state space of the system is denoted by* $\mathcal{X} = L \times X$ *and a state is denoted by* $(l, x) \in \mathcal{X}$*;*
- $E \subseteq L \times L$ *is a set of discrete transitions;*
- $G : E \mapsto 2^X$ *is a guard mapping over discrete transitions;*
- $R : E \times X \mapsto 2^X$ *is a reset mapping over discrete transitions;*
- $I : L \mapsto 2^X$ *is an invariant mapping;*
- $F : L \mapsto (X \mapsto X)$ *is a vector field mapping which assigns to each location* l *a vector field* f_l*.*

The transition and dynamic structure of the hybrid system defines a set of trajectories. A trajectory is a sequence starting from a state $(l_0, x_0) \in \mathcal{X}_0$, where $\mathcal{X}_0 \subseteq \mathcal{X}$ is an initial set, and consisting of a series of interleaved continuous flows and discrete transitions. During the continuous flows, the system evolves following the vector field $F(l)$ at some location $l \in L$ until the invariant condition $I(l)$ is violated. At some state (l, x), if there is a discrete transition $(l, l') \in E$ such that $(l, x) \in G(l, l')$ (we write $G(l, l')$ for $G((l, l')))$, then the discrete transition can be taken and the system state can be reset to $R(l, l', x)$. The problem of safety verification of a hybrid system is to prove that the hybrid system cannot reach an unsafe set $\mathcal{X}_u$ from an initial set $\mathcal{X}_0$.

Some notations that are used in this paper are presented here. $\mathbb{R}$ denotes the real number field. $\mathcal{C}^1(\mathbb{R}^n)$ denotes the space of 1-time continuously differentiable functions mapping $X \subseteq \mathbb{R}^n$ to $\mathbb{R}$. $\mathbb{R}[x]$ denotes the polynomial ring in x over the real number field and $\mathbb{R}[x]^m$ denotes the m-dimensional polynomial vector space over $\mathbb{R}[x]$. M^T denotes the transpose of the matrix M.

3 Conditions for Constructing Barrier Certificates

3.1 Barrier Certificate Condition for Continuous Systems

Given a continuous system S, an initial set X_0 and an unsafe set X_u, a barrier certificate is a real-valued function $\varphi(x)$ of states satisfying that $\varphi(x) \leq 0$ for any point x in the reachable set R and $\varphi(x) > 0$ for any point x in the unsafe set X_u (called *General Constraint* hereafter). Therefore, if there exists such a function $\varphi(x)$, we can assert that $R \cap X_u = \emptyset$, that is, the system can not reach a state in the unsafe set from the initial set. However, the exact reachable set R is not computable for most hybrid systems, we cannot decide directly whether $\varphi(x) \leq 0$ holds for all the points in R. Therefore, various alternative inductive conditions that are equivalent to or sufficient for *General Constraint* were proposed. In what follows, we present a new barrier certificate which is a sufficient condition for *General Constraint*.

Consider a continuous system $\mathbb{C}$ specified by the differential equation (1), we assume that $X_0 (\subseteq X)$, X_u are the initial set and the unsafe set respectively. Then, we have the following theorem as a barrier certificate condition.

Theorem 1 (Exponential Condition). *Given the continuous system* (1) *and the corresponding sets X, X_0 and X_u, for any given $\lambda \in \mathbb{R}$, if there exists a barrier certificate, i.e, a real-valued function $\varphi(x) \in \mathcal{C}^1(\mathbb{R}^n)$ satisfying the following formulae:*

$$\forall x \in X_0 : \varphi(x) \leq 0 \tag{2}$$

$$\forall x \in X : \mathcal{L}_f\varphi(x) - \lambda\varphi(x) \leq 0 \tag{3}$$

$$\forall x \in X_u : \varphi(x) > 0 \tag{4}$$

then the safety property is satisfied by the system (1).

Proof. Suppose $x_0 \in X_0$ and $x(t)$ be the corresponding particular solution of the system (1). We aim to prove that for any function $\varphi(x(t))$ satisfying the formulae (2)- (4), the following formula holds:

$$\forall \zeta \geq 0 : \varphi(x(\zeta)) \leq 0. \tag{5}$$

Let $g(x) = \mathcal{L}_f\varphi(x) - \lambda\varphi(x)$, then by (3)

$$\forall x \in X : g(x) \leq 0 \tag{6}$$

Since $\frac{d\varphi(x(t))}{dt} = \frac{\partial\varphi}{\partial x}\frac{dx}{dt} = \frac{\partial\varphi}{\partial x}f(x) = \mathcal{L}_f\varphi(x)$, we have the differential equation about $\varphi(x(t))$

$$\begin{cases} \frac{d\varphi(x(t))}{dt} - \lambda\varphi(x(t)) - g(x(t)) = 0 \\ \varphi(x(0)) = \varphi(x_0) \end{cases} \tag{7}$$

By solving the differential equation (7), we have the following solution:

$$\varphi(x(t)) = (\int_0^t (g(x(\tau))e^{-\lambda\tau}d\tau + \varphi(x_0))e^{\lambda t}. \tag{8}$$

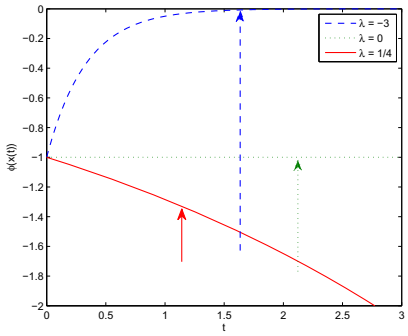


Fig. 1. Dependency of Barrier Certificate Condition on λ. As the value of λ decreases (e.g. from 1/4 to -3), the upper-bound of the value of $\varphi(x(t))$ approaches to zero infinitely, which means the barrier certificate condition becomes less conservative.

By (6), we have

$$\int_0^t (g(x(\tau))e^{-\lambda\tau}d\tau \leq 0. \tag{9}$$

then by (9) and $\varphi(x_0) \leq 0$, we finally have

$$\varphi(x(t)) \leq \varphi(x_0)e^{\lambda t} \leq 0. \tag{10}$$

Hence, for any $\zeta \geq 0$, $\varphi(x(\zeta)) \leq 0$ holds. □

Remark 1. The formulae (2) and (4) ensure that the barrier separates the initial set X_0 from the unsafe set X_u, and the formula (3) ensures that system trajectories cannot escape from inside of the barrier. These formulae together imply that $\varphi(x) \leq 0$ is an inductive invariant of the system (1).

From another point of view, the semi-algebraic set $\{x \in \mathbb{R}^n | \varphi(x) \leq 0\}$ forms an over-approximation for the reachable set of the system (1), and the zero level set of the function $\varphi(x)$ (i.e., $\{x \in \mathbb{R}^n | \varphi(x) = 0\}$) forms the boundary of the over-approximation. In order to be less conservative, we hope the boundary of the over-approximation encloses the reachable set $\{x(t) | x(0) \in X_0, \dot{x} = f(x), t \in \mathbb{R}_+\}$ as tightly as possible, in other words, to make the upper-bound of $\varphi(x(t))$ approach zero as closely as possible. According to the above proof (i.e., (10)), the scope over which the function $\varphi(x(t))$ can range depends closely on the value of the parameter λ: the less value the λ is, the closer the upper-bound of the scope that $\varphi(x(t))$ can reach is to zero (see Fig. 1). Roughly speaking, the values of λ are divided into three classes according to the conservativeness of the barrier certificate condition:

- $\lambda = 0$. In this case, the formula (3) is degenerated to $\frac{\partial \varphi}{\partial x} f(x) \leq 0$, which is the case of *Convex Condition*. This condition implies that the value of $\varphi(x(t))$ will never get close to zero over time t. Thus, the condition is very conservative. Similarly, *Differential Invariant* is a generalization of *Convex Condition* and accordingly it completely inherits the conservativeness of *Convex Condition* (Refer to [18] for a detailed explanation on this point).
- $\lambda < 0$. In this case, we know that 1) $\varphi(x(t)) \leq \varphi(x_0)e^{\lambda t} \leq 0$, and 2) $\frac{\partial \varphi}{\partial x} f(x) \leq \lambda \varphi(x) \geq 0$. These two inequalities together imply that the value of $\varphi(x(t))$ can increase over the time t but never get across the upper bound 0, provided that $\varphi(x(0)) \leq 0$ at the beginning.
- $\lambda > 0$. In this case, $\frac{\partial \varphi}{\partial x} f(x) \leq \lambda \varphi(x) \leq 0$, which means that the value of $\varphi(x(t))$ get far away from 0. Apparently, the condition is much more conservative than the first case.

Therefore, as long as we let $\lambda < 0$, we can get less conservative barrier certificate conditions than *Convex Condition* and *Differential Invariant*. Note that *Exponential Condition* is convex as well and its convexity can be easily proved by verifying that for any two functions $\varphi_1(x)$ and $\varphi_2(x)$ satisfying the formulae (2)–(4) and any θ with $0 \leq \theta \leq 1$, $\varphi(x) = \theta\varphi_1(x) + (1-\theta)\varphi_2(x)$ satisfies the formulae (2)–(4) as well. Based on this fact, we can convert the problem of constructing barrier certificate into the problem of convex optimization which we will discuss in Section 4.

In the following subsection, we extend the barrier certificate condition for continuous systems to hybrid systems.

3.2 Barrier Certificate Condition for Hybrid Systems

Different from the barrier certificate for a continuous system, the barrier certificate for a hybrid system consists of a set of functions $\{\varphi_l(x) | l \in L\}$, each of which corresponds to a discrete location of the system and forms a barrier between the reachable set and the unsafe set at that individual location. For each function $\varphi_l(x)$ at location l, in addition to defining constraints for the continuous flows, the barrier certificate conditions have to take into account all the discrete transitions starting from location l to make the overall barrier certificate an inductive invariant. Formally, we define the barrier certificate condition for hybrid systems as the following theorem.

Theorem 2 (Hybrid-Exp Condition). *Given the hybrid system* $\mathcal{H} = \langle L, X, E, R, G, I, F \rangle$, *the initial set* $\mathcal{X}_0$ *and the unsafe set* $\mathcal{X}_u$ *of* $\mathcal{H}$, *then, for any given set of constant real numbers* $S_\lambda = \{\lambda_l \in \mathbb{R} | l \in L\}$ *and any given set of constant non-negative real numbers* $S_\gamma = \{\gamma_{ll'} \in \mathbb{R}_+ | (l, l') \in E\}$, *if there exists a set of functions* $\{\varphi_l(x) | \varphi_l(x) \in \mathcal{C}^1(\mathbb{R}^n), l \in L\}$ *such that, for all* $l \in L$ *and* $(l, l') \in E$, *the following formulae hold:*

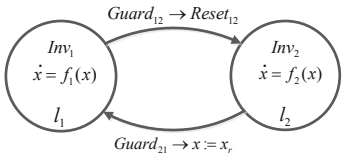


Fig. 2. A hybrid system without barrier certificate satisfying *Convex Condition*

$$\forall x \in \mathrm{Init}(l) : \varphi_l(x) \leq 0 \tag{11}$$
$$\forall x \in I(l) : \mathcal{L}_{f_l}\varphi_l(x) - \lambda_l\varphi_l(x) \leq 0 \tag{12}$$
$$\forall x \in G(l,l'), \forall x' \in R((l,l'),x) : \gamma_{ll'}\varphi_l(x) - \varphi_{l'}(x') \geq 0 \tag{13}$$
$$\forall x \in \mathrm{Unsafe}(l) : \varphi_l(x) > 0 \tag{14}$$

where $\mathrm{Init}(l)$ *and* $\mathrm{Unsafe}(l)$ *denote respectively the initial set and the unsafe set at location* l*, then the safety property is satisfied by* $\mathcal{H}$*.*

The proof of Theorem 2 can be found in [18]. Informally, the formulae (11), (12) and (14) together ensure that at each location $l \in L$, the system never evolves into an unsafe state continuously. The formula (13) ensures that the system never jumps from a safe state to an unsafe state discretely. By induction, the formulae (11)–(14) together guarantee the safety of the system.

Remark 2. The selection of the parameter set S_λ is essential to the conservativeness of the barrier certificate conditions. As discussed in Subsection 3.1, by setting all the elements of S_λ to 0, we can derive *Convex Condition* for hybrid systems. However, *Convex Condition* is too restrictive to be useful for hybrid systems. For example, see the hybrid system in Fig. 2, there is a reset operation $x = x_r$ (which is often the case) at the transition (l_2, l_1). Assume there exists a barrier certificate $\{\varphi_{l_1}(x), \varphi_{l_2}(x)\}$ if we set all the elements of S_λ to 0 and (without loss of generality) set all the elements of S_γ to 1, then for any trajectory containing at least two times of the transition (l_2, l_1), one at time instant t_1 and another at t_2, $t_1 < t_2$, respectively, we can assert that $\varphi_{l_1}(x_{l_1t_1}) > \varphi_{l_1}(x_{l_1t_2})$ according to Theorem 2, this contradicts with $x_{l_1t_1} = x_{l_1t_2} = x_r$, that is, the barrier certificate satisfying *Convex Condition* does not exist no matter what the unsafe set is. Therefore, in order to make the barrier certificate condition less conservative, we try to choose negative values for $\lambda_l \in S_\lambda$ and theoretically: the less, the better. However, in practice, the optimal domain for λ may depend on the specific computational method. For example, the interval $[-1, 0)$ appears to be optimal and not too sensitive in-between for the semidefinite programming method used in this paper.

The selection of S_γ is relatively simple. We usually set all of its elements to 1 except for the discrete jumps with a reset operation that is independent of the pre-state of the jump, for which we usually set $\gamma_{ll'}$ to 0.

4 Construction Method for Barrier Certificate

Constructing inductive invariants for general hybrid systems is very hard. Fortunately, for some existing inductive conditions, several computational methods are available for semialgebraic hybrid systems. The most representative methods include the fixed-point method based on saturation [11], the constraint-solving methods based on semidefinite programming [9] and quantifier elimination [12] and the Gröbner bases method [7], [6]. Similar to *Convex Condition*, *Exponential Condition* defines a convex set of barrier certificate functions as well and hence can be solved by semidefinite programming method supposing the hybrid system is semialgebraic and the barrier certificate function $\varphi(x)$ is a polynomial.

In our computational method, a barrier certificate is assumed to be a set $\Phi = \{\varphi_l(x)|l \in L\}$ of multivariate polynomials of fixed degrees with a set of unknown real coefficients. According to the constraint inequalities in Theorem 1 or Theorem 2, we can obtain a set of positive semidefinite (*PSD*) polynomials $Q = \{Q_i|Q_i(x) \geq 0, deg(Q_i) = 2n, x \in \mathbb{R}^n, n \in \mathbb{N}\}$, where $deg(\cdot)$ returns the degree of a polynomial. Note that a polynomial $Q(x)$ of degree $2k$ is said to be *PSD if and only if* $Q(x) \geq 0$ for all $x \in \mathbb{R}^n$. Thus, our objective is to find a set of real-valued coefficients for $\varphi_l \in \Phi$ to make all the $Q_i \in Q$ be *PSD*.

A famous sufficient condition for a polynomial $P(x)$ of degree $2k$ to be *PSD* is that it is a sum-of-squares (*SOS*) $P(x) = \sum q_i(x)^2$ for some polynomials $q_i(x)$ of degree k or less [19]. Furthermore, it is equivalent to that $P(x)$ has a positive semidefinite quadratic form, i.e., $P(x) = v(x)Mv(x)^T$, where $v(x)$ is a vector of monomials with respect to x of degree k or less and M is a real symmetric *PSD* matrix with the coefficients of $P(x)$ as its entries. Therefore, the problem of finding a *PSD* polynomial $P(x)$ can be converted to the problem of solving a linear matrix inequality (*LMI*) $M \succeq 0$ [20], which can be solved by semidefinite programming [21].

In our work, we extend SOSTOOLS based on the theory in this paper to implement an algorithm for discovering barrier certificate automatically.

4.1 Sum-of-Squares Transformation for Continuous System

In order to be solvable for the barrier certificate condition by *SOS* programming, we need to restate it with multivariate polynomials. In this context, we assume that all the state sets involved in the condition are semialgebraic, that is, they can be written as $\{x \in \mathbb{R}^n|P_1(x) \geq 0, ..., P_m(x) \geq 0, P_i(x) \in \mathbb{R}[x], 1 \leq i \leq m\}$). For convenience, we write it compactly as $\{x \in \mathbb{R}^n|\mathcal{P}(x) \geq 0, \mathcal{P}(x) \in \mathbb{R}[x]^m\}$, where $\mathcal{P}(x) = (P_1(x), P_2(x), ..., P_m(x))$. In addition, each dimension of the vector field $f(x)$ and the barrier certificate function $\varphi(x)$ are all polynomials in $\mathbb{R}[x]$. Based on the previous assumption, we present the sum-of-squares transformation of *Exponential Condition* for continuous systems as the following corollary.

Corollary 1. *Given the continuous polynomial system* (1) *and the initial set* $X_0 = \{x \in \mathbb{R}^n|I_0(x) \geq 0, I_0(x) \in \mathbb{R}[x]^r\}$ *and the unsafe set* $X_u = \{x \in \mathbb{R}^n|U(x) \geq 0, U(x) \in \mathbb{R}[x]^s\}$, *where* r *and* s *are the dimensions of the polynomial vector spaces, for any given* $\lambda \in \mathbb{R}$ *and any given real number* $\epsilon > 0$,

if there exists a polynomial function $\varphi(x) \in \mathbb{R}[x]$ *and two* SOS *polynomial vectors (i.e., every element of the vector is a* SOS *polynomial)* $\mu(x) \in \mathbb{R}[x]^r$ *and* $\eta(x) \in \mathbb{R}[x]^s$ *satisfying that the following polynomials*

$$-\varphi(x) - \mu(x)I_0(x) \tag{15}$$

$$-\mathcal{L}_f\varphi(x) + \lambda\varphi(x) \tag{16}$$

$$\varphi(x) - \eta(x)U(x) - \epsilon \tag{17}$$

are all SOS*s, then the safety property is satisfied by the system* (1).

Proof. It is sufficient to prove that any $\varphi(x)$ satisfying (15)–(17) also satisfies (2)–(4). By (15), we have $-\varphi(x) - \mu(x)I_0(x) \geq 0$, that is, $\varphi(x) \leq -\mu(x)I_0(x)$. Because for any $x \in X_0$, $-\mu(x)I_0(x) \leq 0$, this means $\varphi(x) \leq 0$. Similarly, we can derive (3) from (16). By (17), it's easy to prove that $\varphi(x) - \epsilon \geq 0$ holds for any $x \in X_u$. Since ϵ is greater than 0, then the formula (4) holds. Therefore, the system (1) is safe. □

Remark 3. Since the polynomials (15)–(17) are required to be *SOS*s, each of them can be transformed to a positive semidefinite quadratic form $v(x)M_iv(x)^T$, where M_i is a real symmetric *PSD* matrix with the coefficients of $\varphi(x)$, $\mu(x)$ and $\eta(x)$ as its variables. As a result, we obtain a set of *LMI*s $\{M_i \succeq 0\}$ which can be solved by semidefinite programming.

We use *Algorithm* 1 to compute the desired barrier certificate. In the algorithm, we first choose a small set of negative values Λ as a candidate set for λ and an integer interval $[dMin, dMax]$ as a candidate set for degree d of $\varphi(x)$. Then, we attempt to find a barrier certificate satisfying the formulae (15)–(17) for a fixed pair of λ and d until such one is found. Theoretically, according to the analysis about the dependence of conservativeness of barrier certificate on the value of λ, we should set λ to as small negative value as possible. However, experiments show that too small negative numbers for λ often lead the semidefinite programming function to numerical problems. In practice, the negative values in the interval $[-1, 0)$ are good enough for λ to verify very critical safety properties. Note that the principle for step 3 in *Algorithm 1* is that if $\varphi(x)$ has a dominating degree in both polynomials, there couldn't exist a solution that make both polynomials be *SOS*s because $-\varphi(x)$ and $\varphi(x)$ occur in (15) and (17) simultaneously. The motive for eliminating the monomials with small coefficients in step 7 is from the observation that those monomials are usually the cause of the failed *SOS* decomposition for the polynomials when the semidefinite programming function gives a seemingly feasible solution.

The idea for constructing barrier certificates for continuous systems can be easily extended to hybrid systems. We describe it in the following subsection.

4.2 Sum-of-Squares Transformation for Hybrid System

Similar to continuous system, in order to be solvable by semidefinite programming, we need to limit the hybrid system model in Section 2 to semialgebraic hybrid system.

Algorithm 1. Computing Barrier Certificate for Continuous System

Input: f: array of polynomial vector field; I_0: array of polynomials defining X_0; U: array of polynomials defining X_u
Output: φ: barrier certificate polynomial
Variables : λ: a real negative value; d: degree of φ
Constants: Λ: array of candidate values for λ; ϵ: a positive value; $dMin$, $dMax$: the minimal degree and maximal degree of φ to be found

1 Initialize. Set Λ to a set of negative values between -1 and 0; Set ϵ to a small positive value; Set $dMin$ and $dMax$ to positive integer respectively;
2 Pick λ and d. For each $\lambda \in \Lambda$ and for each d from $dMin$ to $dMax$, perform step 3–7 until a barrier certificate is found;
3 Decide the degree of $\mu(x)$ and $\eta(x)$ according to d. To be SOSs for both (15) and (17), at least one of the degrees of $\mu(x)I_0(x)$ and $\eta(x)U(x)$ is greater than or equal to the degree of $\varphi(x)$;
4 Generate complete polynomials $\varphi(x)$, $\mu(x)$ and $\eta(x)$ of specified degree with unknown coefficient variables;
5 Eliminate the monomials of odd top degrees in (15)–(17), $\mu(x)$ and $\eta(x)$, respectively. To be a SOS, a polynomial has to be of even degree. Concretely, let the coefficients of the monomials to be eliminated be zero to get equations about coefficient variables and then reduce the number of coefficient variables by solving the equations and substituting free variables for non-free variables in all the related polynomials;
6 Perform the SOS programming on the positive semidefinite constraints (15)–(17) and $\mu(x)$, $\eta(x)$;
7 Check if a feasible solution is found, if not found, continue with a new loop; else, check if the solution can indeed enable the corresponding polynomials to be SOSs, if so, return $\varphi(x)$; else, for all the polynomials in the programming, eliminate all the monomials whose coefficients have too small absolute values(usually less than 10^{-5}) by using the same method as step 5, then go to step 6 unless an empty polynomial is produced;

Consider the hybrid system $\mathbb{H} = \langle L, X, E, R, G, I, F\rangle$, where the mappings F, R, G, I of $\mathbb{H}$ are defined with respect to polynomial inequalities as follows:

- $F : l \mapsto\ f_l(x)$
- $G : (l, l') \mapsto \{x \in \mathbb{R}^n | G_{ll'}(x) \geq\ 0, G_{ll'}(x) \in \mathbb{R}[x]^{p_{ll'}}\}$
- $R : (l, l', x) \mapsto\ \{x' \in \mathbb{R}^n | R_{ll'x}(x') \geq\ 0, R_{ll'x}(x') \in \mathbb{R}[x]^{q_{ll'}}\}$
- $I : l \mapsto \{x \in \mathbb{R}^n | I_l(x) \geq\ 0, I_l(x) \in \mathbb{R}[x]^{r_l}\}$

and the mappings of the initial set and the unsafe set are defined as follows:

- $\text{Init} : l \mapsto \{x \in \mathbb{R}^n | \text{Init}_l(x) \geq 0, \text{Init}_l(x) \in \mathbb{R}[x]^{s_l}\}$
- $\text{Unsafe} : l \mapsto \{x \in \mathbb{R}^n | \text{Unsafe}_l(x) \geq 0, \text{Unsafe}_l(x) \in \mathbb{R}[x]^{t_l}\}$

where $p_{ll'}$, $q_{ll'}$, r_l, s_l and t_l are the dimensions of polynomial vector spaces. Then we have the following corollary for constructing barrier certificate for the semialgebraic hybrid system $\mathbb{H}$.

Corollary 2. *Let the hybrid system* $\mathbb{H}$ *and the initial state set mapping* Init *and the unsafe state set mapping* Unsafe *be defined as the above. Then, for any given set of constant real numbers* $S_\lambda = \{\lambda_l \in \mathbb{R} | l \in L\}$ *and any given set of constant non-negative real numbers* $S_\gamma = \{\gamma_{ll'} \in \mathbb{R}_+ | (l,l') \in E\}$ *,and any given small real number* $\epsilon > 0$*, if there exists a set of polynomial functions* $\{\varphi_l(x) \in \mathbb{R}[x] | l \in L\}$ *and five sets of* SOS *polynomial vectors* $\{\mu_l(x) \in \mathbb{R}[x]^{s_l} | l \in L\}$, $\{\theta_l(x) \in \mathbb{R}[x]^{r_l} | l \in L\}$, $\{\kappa_{ll'}(x) \in \mathbb{R}[x]^{p_{ll'}} | (l,l') \in E\}$, $\{\sigma_{ll'}(x) \in \mathbb{R}[x]^{q_{ll'}} | (l,l') \in E\}$ *and* $\{\eta_l(x) \in \mathbb{R}[x]^{t_l} | l \in L\}$*, such that the polynomials*

$$\varphi_l(x) - \mu_l(x)\,\mathrm{Init}_l(x) \tag{18}$$

$$\lambda_l \varphi_l(x) - \mathcal{L}_{f_l}\varphi_l(x) - \theta_l(x) I_l(x) \tag{19}$$

$$\gamma_{ll'}\varphi_l(x) - \varphi_{l'}(x') - \kappa_{ll'}(x) G_{ll'}(x) - \sigma_{ll'}(x') R_{ll'x}(x') \tag{20}$$

$$\varphi_l(x) - \epsilon - \eta_l(x)\,\mathrm{Unsafe}_l(x) \tag{21}$$

are SOS*s for all* $l \in L$ *and* $(l,l') \in E$*, then the safety property is satisfied by the system* $\mathbb{H}$*.*

Proof. Similar to Corollary 1, it's easy to prove that any set of polynomials $\{\varphi_l(x)\}$ satisfying (18)–(21) also satisfies (11)–(14), hence the hybrid system $\mathbb{H}$ is safe. □

The algorithm for computing the barrier certificates for hybrid systems is similar to the algorithm for continuous systems except that it needs to take into account the constraint (20) for the discrete transitions. We do not elaborate on it here any more. Note that the strategy for the selection of λ's for continuous system applies here as well and we only need to set all the elements of S_γ to 1 except for the discrete transition whose post-state is independent of the pre-state, where we set $\gamma_{ll'}$ to 0 to reduce the computational complexity.

5 Examples

5.1 Example 1

Consider the two-dimensional system (from [22] page 315)

$$\begin{bmatrix} \dot{x_1} \\ \dot{x_2} \end{bmatrix} = \begin{bmatrix} x_2 \\ -x_1 + \frac{1}{3}x_1^3 - x_2 \end{bmatrix}$$

with $\mathcal{X} = \mathbb{R}^2$, we want to verify that starting from the initial set $X_0 = \{x \in \mathbb{R}^2 | (x_1 - 1.5)^2 + x_2^2 \leq 0.25\}$, the system will never evolve into the unsafe set $X_u = \{x \in \mathbb{R}^2 | (x_1+1)^2 + (x_2+1)^2 \leq 0.16\}$. We attempted to use both the *Convex-Condition*-based method proposed in [8] and the *Exponential-Condition*-based method in this paper to find the barrier certificates with a degree ranging from 2 to 10. (Note that in [12], [13], the inductive invariants are not sufficient in general according to [14] and hence cannot be applied to our examples. The work of [19] applies only to a very special class of hybrid systems which is not applicable

to our examples either.) During this process, all the programming polynomials are complete polynomials automatically generated (instead of the non-complete polynomials consisting of painstakingly chosen terms) and all the computations are performed in the same environment. The result of the experiment is listed in Table 5.1. The first column is the degree of the barrier certificate to be found, the second column is the runtime spent by the *Convex-Condition*-based method, and the rest columns are the runtime spent by the *Exponential-Condition*-based method for different value of λ. Note that the symbol $\times$ in the table indicates that the method failed to find a barrier certificate with the corresponding degree either because the semidefinite programming function found no feasible solution or because it ran into a numerical problem.

Table 1. Computing results for *Convex Condition* and *Exponential Condition*

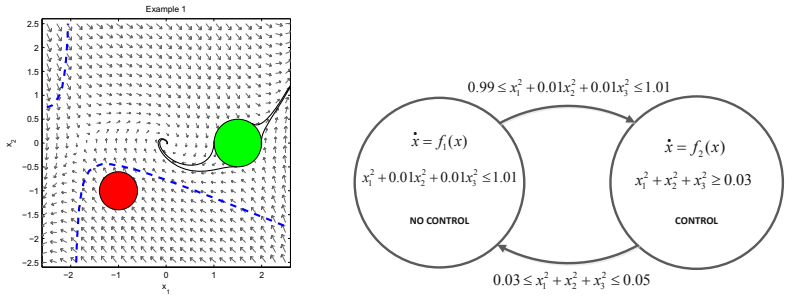
Degree of $\varphi(x)$	Convex Condition	Exponential Condition		
	$Time$(sec)	$Time$(sec)		
		$\lambda = \frac{-1}{8}$	$\lambda = \frac{-1}{4}$	$\lambda = -1$
2	$\times$	0.4867	0.4836	0.2496
3	$\times$	0.5444	0.6224	0.4976
4	0.4368	0.4103	0.4072	0.3853
5	$\times$	0.4321	0.4103	0.3947
6	$\times$	0.3214	0.3011	0.2714
7	$\times$	0.9563	0.9532	0.9453
8	$\times$	0.9188	0.8970	0.7893
9	$\times$	1.4944	1.4149	1.5132
10	$\times$	1.4336	1.3931	1.3650

As shown in Table 5.1, the *Convex-Condition*-based method succeeded only in one case ($Degree = 4$) due to the conservativeness of *Convex Condition*. Comparably, our method found all the barrier certificates of the specified degrees ranging from 2 to 10. Especially, the lowest degree of barrier certificate we found is quadratic: $\varphi(x) = -.86153 - .87278x_1 - 1.1358x_2 - .23944x_1^2 - .5866x_1x_2$ with $\mu(x) = 0.75965$ and $\eta(x) = 0.73845$ when λ is set to -1. The phase portrait of the system and the zero level set of $\varphi(x)$ are shown in Fig. 3(a). Note that being able to find a lower degree of barrier certificates is essential in reducing the computational complexity.

In addition, we can see from Table 5.1 that the runtime of *Exponential-Condition*-based method decreases with the value of λ for each fixed degree except for $Degree = 3, 9$, this observation can greatly evidence our theoretical result about λ selection: the less, the better.

5.2 Example 2

In this example, we consider a hybrid system with two discrete locations (from [9]). The discrete transition diagram of the system is shown in Fig. 3(b) and the vector fields describing the continuous behaviors are as follows:



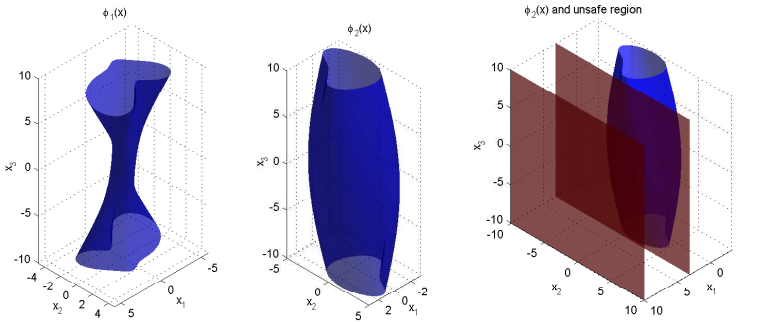
(a) Subsection 5.1 (b) Subsection 5.2

Fig. 3. (a) Phase portrait of the system in Subsection 5.1. The solid patches from right to left are X_0 and X_u, respectively, the solid lines depict the boundary of the reachable region of the system from X_0, and the dashed lines are the zero level set of a quadratic barrier certificate $\varphi(x)$ which separates the unsafe region X_u from the reachable region. (b) Discrete transition diagram of the hybrid system in Subsection 5.2.

$$f_1(x) = \begin{bmatrix} x_2 \\ -x_1 + x_3 \\ x_1 + (2x_2 + 3x_3)(1 + x_3^2) \end{bmatrix}, f_2(x) = \begin{bmatrix} x_2 \\ -x_1 + x_3 \\ -x_1 - 2x_2 - 3x_3 \end{bmatrix}$$

At the beginning, the system is initialized at some point in $X_0 = \{x \in \mathbb{R}^3 | x_1^2 + x_2^2 + x_3^2 \le 0.01\}$ and then it starts to evolve following the vector fields $f_1(x)$ at location 1(NO CONTROL mode). When the system reaches some point in the guard set $G(1,2) = \{x \in \mathbb{R}^3 | 0.99 \le x_1^2 + 0.01x_2^2 + 0.01x_3^3 \le 1.01\}$, it can jump to location 2 (CONTROL mode) nondeterministically without performing any reset operation (i.e., $R(1,2,x) = G(1,2)$). At location 2, the system will operate following the vector field $f_2(x)$, which means that a controller will take over to prevent x_1 from getting too big. As the system enters the guard set $G(2,1) = \{x \in \mathbb{R}^3 | 0.03 \le x_1^2 + x_2^2 + x_3^2 \le 0.05\}$, it will jump back to location 1 nondeterministically again without reset operation (i.e., $R(2,1,x) = G(2,1)$). Different from the experiment in [9], where the objective is to verify that $|x_1| < 5.0$ in CONTROL mode, our objective is to verify that x_1 will stay in a much more restrictive domain in CONTROL mode: $|x_1| < 3.2$.

We define the unsafe set as $\text{Unsafe}(1) = \emptyset$ and $\text{Unsafe}(2) = \{x \in \mathbb{R}^3 | 3.2 \le x_1 \le 10\} \cup \{x \in \mathbb{R}^3 | -10 \le x_1 \le -3.2\}$, which is sufficient to prove $|x_1| \le 3.2$ in CONTROL mode. Similarly, we tried to use both the method in this paper and the method in [8] to compute the barrier certificate. By setting $\lambda_1 = \lambda_2 = -\frac{1}{5}$ and $\gamma_{12} = \gamma_{21} = 1$, our method found a pair of quartic barrier certificate functions: $\phi_1(x)$ and $\phi_2(x)$, whose zero level set is shown in Fig. 4(a) and Fig. 4(b) respectively. As you can see, at each location $l = 1, 2$, the zero level set of $\phi_l(x)$ forms the boundary of the over-approximation $\phi_l(x) \le 0$ (denoting the points within the pipe) for the reachable set at location l. On the one hand, the hybrid system starts from and evolves within the corresponding over-approximation and jumps back and forth between the two over-approximations. On the other hand,



(a) $\phi_1(x) = 0$ (b) $\phi_2(x) = 0$ (c) $3.2 \leq x_1 \leq 10$, $\phi_2(x) = 0$

Fig. 4. Barrier certificates $\phi_1(x)$ and $\phi_2(x)$ for the hybrid system in Subsection 5.2. $\phi_l(x) = 0$ $(l = 1, 2)$ forms the boundary of the over-approximation $\phi_l(x) \leq 0$ and separates the inside reachable set from the outside unsafe set (e.g. $3.2 \leq x_1 \leq 10$).

the unsafe set does not intersect the over-approximation formed by $\phi_2(x) \leq 0$ (see Fig. 4(c)). Therefore, the safety of the system is guaranteed. However, using the method in [8], we cannot compute the barrier certificate, which means it cannot verify the system.

6 Conclusion

In this paper, we propose a new barrier certificate condition (called *Exponential Condition*) for the safety verification of hybrid systems. Our barrier certificate condition is parameterized by a real number λ and the conservativeness of the barrier certificate condition depends closely on the value of λ: the less value the λ is, the less conservative the barrier certificate condition is. The most important benefit of *Exponential Condition* is that it possesses a relatively low conservativeness as well as the convexity and hence can be solved efficiently by semidefinite programming method.

Based on our method, we are able to construct polynomial barrier certificate to verify very critical safety property for semialgebraic continuous systems and hybrid systems. The experiments on a continuous system and a hybrid system show the effectiveness and practicality of our method.

References

1. Henzinger, T.: The theory of hybrid automata. In: Proc. IEEE Symp. Logic in Computer Science (LICS), pp. 278–292 (1996)
2. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T., Ho, P., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
3. Jirstrand, M.: Invariant sets for a class of hybrid systems. In: Proc. IEEE Conference on Decision and Control, vol. 4, pp. 3699–3704 (1998)

4. Rodríguez-Carbonell, E., Tiwari, A.: Generating polynomial invariants for hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 590–605. Springer, Heidelberg (2005)
5. Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Constructing invariants for hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 539–554. Springer, Heidelberg (2004)
6. Sankaranarayanan, S.: Automatic invariant generation for hybrid systems using ideal fixed points. In: Proc. ACM International Conference on Hybrid Systems: Computation and Control, pp. 221–230 (2010)
7. Tiwari, A., Khanna, G.: Nonlinear systems: Approximating reach sets. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 600–614. Springer, Heidelberg (2004)
8. Prajna, S., Jadbabaie, A., Pappas, G.: A framework for worst-case and stochastic safety verification using barrier certificates. IEEE Transactions on Automatic Control 52(8), 1415–1428 (2007)
9. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
10. Sloth, C., Pappas, G., Wisniewski, R.: Compositional safety analysis using barrier certificates. In: Proc. ACM International Conference on Hybrid Systems: Computation and Control, pp. 15–24 (2012)
11. Platzer, A., Clarke, E.M.: Computing differential invariants of hybrid systems as fixedpoints. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 176–189. Springer, Heidelberg (2008)
12. Gulwani, S., Tiwari, A.: Constraint-based approach for analysis of hybrid systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 190–203. Springer, Heidelberg (2008)
13. Taly, A., Tiwari, A.: Deductive verification of continuous dynamical systems. In: FSTTCS, vol. 4, pp. 383–394 (2009)
14. Taly, A., Gulwani, S., Tiwari, A.: Synthesizing switching logic using constraint solving. Intl. J. Software Tools for Technology Transfer 13(6), 519–535 (2011)
15. Prajna, S., Papachristodoulou, A., Seiler, P., Parrilo, P.: SOSTOOLS and its control applications. Positive Polynomials in Control, pp. 580–580 (2005)
16. Carloni, L., Passerone, R., Pinto, A.: Languages and tools for hybrid systems design. Foundations and Trends® in Electronic Design Automation 1(1-2) (2006)
17. Maler, O., Manna, Z., Pnueli, A.: Prom timed to hybrid systems. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 447–484. Springer, Heidelberg (1992)
18. Kong, H., He, F., Song, X., Hung, W.N.N., Gu, M.: Exponential-Condition-Based Barrier Certificate Generation for Safety Verification of Hybrid Systems (March 2013), ArXiv e-prints: `http://arxiv.org/abs/1303.6885`
19. Lasserre, J.: Sufficient conditions for a real polynomial to be a sum of squares. Archiv der Mathematik 89(5), 390–398 (2007)
20. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear matrix inequalities in system and control theory. Society for Industrial Mathematics, vol. 15 (1994)
21. Parrilo, P.: Semidefinite programming relaxations for semialgebraic problems. Mathematical Programming 96(2), 293–320 (2003)
22. Khalil, H.K.: Nonlinear Systems, 3rd edn. Prentice Hall (2001)

Flow*: An Analyzer for Non-linear Hybrid Systems

Xin Chen[1], Erika Ábrahám[1], and Sriram Sankaranarayanan[2,⋆]

[1] RWTH Aachen University, Germany
{xin.chen,abraham}@cs.rwth-aachen.de
[2] University of Colorado, Boulder, CO.
srirams@colorado.edu

Abstract. The tool FLOW* performs Taylor model-based flowpipe construction for non-linear (polynomial) hybrid systems. FLOW* combines well-known Taylor model arithmetic techniques for guaranteed approximations of the continuous dynamics in each mode with a combination of approaches for handling mode invariants and discrete transitions. FLOW* supports a wide variety of optimizations including adaptive step sizes, adaptive selection of approximation orders and the heuristic selection of template directions for aggregating flowpipes. This paper describes FLOW* and demonstrates its performance on a series of non-linear continuous and hybrid system benchmarks. Our comparisons show that FLOW* is competitive with other tools.

1 Overview of FLOW*

In this paper, we present the FLOW* tool to generate flowpipes for non-linear hybrid systems using *Taylor Models* (TMs). TMs were originally proposed by Berz and Makino [1] to represent functions by means of higher-order Taylor polynomial expansions, bloated by an interval to represent the approximation error. TMs support functional operations such as addition, multiplication, division, derivation and anti-derivation. Guaranteed integration techniques can utilize TMs to provide tight flowpipe over-approximations to non-linear ODEs, with each flowpipe segment represented by a TM [2]. However, these techniques do not naturally extend to *non-linear* hybrid systems consisting of multiple modes and discrete transitions (jumps).

Figure 1 presents a schematic diagram of the major components of FLOW*. FLOW* accepts (i) A hybrid system model file which describes the modes, the polynomial dynamics associated with each mode and the transitions between modes; (ii) A specification file includes TM flowpipes with the state space and unsafe set specifications. For a model file, FLOW* performs a flowpipe construction for a specified time horizon $[0, T]$ and a maximum jump depth J such that the flowpipe set is an over-approximation of the states which can be reached in $[0, T]$ with at most J jumps. FLOW* also checks whether the flowpipe intersects the unsafe set and outputs a visualization of the set of reachable states using polyhedral over-approximations of the computed TM flowpipes. FLOW* is extensible in quite simple ways. Our TM output can be parsed in by other

⋆ Sankaranarayanan's research is supported by the US National Science Foundation (NSF) under grant numbers CNS-0953941 and CPS-1035845. All opinions expressed are those of the authors and not necessarily of the NSF.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 258–263, 2013.

tools, including FLOW* itself to check multiple properties incrementally. This can help support future advances such as checking for MTL property satisfaction for flowpipes, finding limit cycles, inferring likely invariants or Lyapunov functions from flowpipes and extending robustness metric computations for entire flowpipes [3]. The FLOW* tool with its source code and the set of non-linear benchmarks used in this paper, as well as our original work [4] are available on-line[1].

TM Integrator: The Taylor model integrator implements a guaranteed integration scheme using higher-order TMs, along the lines of previous work described in [1,2,5,6]. Our implementation includes numerous enhancements to the existing TM integration techniques including the adaptive adjustment of the orders of the TMs for better control of integration error and better handling of flowpipe guard intersections.

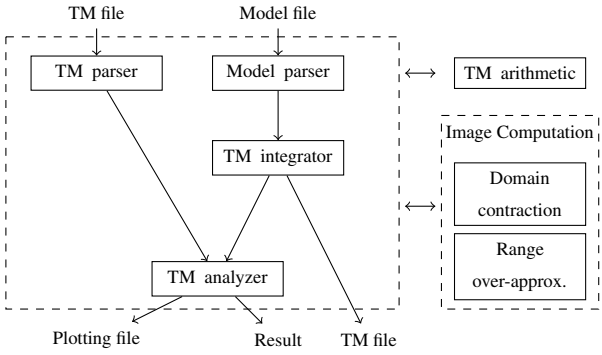


Fig. 1. Structure of FLOW*

Image Computation for Discrete Transitions: The implementation of image computation is based on the techniques described in our previous work for approximating the intersection of TMs with guard sets of the transitions [4]. This approximation is achieved by two complementary techniques: (a) the domain contraction technique computes a smaller TM by shrinking the initial condition and the time interval for which an intersection with the guard is possible; (b) the range over-approximation technique that converts TMs into representations such as template polyhedron or zonotopes over which guard intersection can be computed efficiently. Range over-approximation also includes the conversion of the result back to a TM. FLOW* implements a combination of (a) and (b) to achieve a better accuracy than using either of them.

The verification techniques developed for hybrid systems over the last few decades have resulted in many tools for linear hybrid systems analysis including HyTech, Checkmate, d/dt, Ariadne, HySAT/iSAT, RealPaver, PHAVer, SpaceEx [7]. However, few tools exist for non-linear systems. A few notable non-linear analysis tools include KeYmaera [8], Ariadne [9] and HySAT/iSAT [10].

2 Novel Features in FLOW*

We discuss features of the FLOW* which have not been included in our earlier work [4]. These features improve the efficiency of our overall approach and allow the automatic

[1] `http://systems.cs.colorado.edu/research/cyberphysical/taylormodels/`

Name	Description
Remainder Interval (I_e)	Maximum remainder interval I_e for each integration step.
	Adapt step sizes and TM orders to ensure error within I_e.
Adaptive Step Sizes	Range $[\alpha, \beta]$ for possible step sizes.
Adaptive TM Orders	Change TM orders on-the-fly.
	Allow state variables to have different orders.
Preconditioning	Change of basis for better flow approximation.
Template Directions	Aggregating flowpipe segments by a template polyhedron.

Fig. 2. Basic parameters for controlling FLOW* algorithm

selection of parameters. In our experience, the integration of non-linear systems is often quite *fragile*. There are many parameters to adjust, as outlined in Figure 2. It is necessary to choose them judiciously to ensure that the desired flowpipe accuracy is maintained without expending too much resources. Automating the choice of some of these parameters on-the-fly seems to be the only possible solution to this problem. The new features added to FLOW* automate this to a large extent by allowing the user to specify a flexible range of parameters and adapting the flowpipe construction on the fly within this range to trade off precision of the result against the running time and memory consumption.

Specifying Different Orders over Dimensions. The performance and accuracy of TM integration depends critically on the order of the TM approximation chosen. However, existing approaches specify a single fixed order for each state variable. Nevertheless, it is clear that for a complex system, different state variables grow at varying rates. A key feature of FLOW* allows us to perform integration while specifying orders for the TM representation of each state variable independently during the integration process. For instance, state variables that represent timers can be specified to have order 1 TMs, whereas fast varying variables can be represented by higher order TMs at the same time.

Adaptive Techniques. FLOW* supports adaptive integration time step and adaptive TM order selection for each state variable using specially designed schemes.
Adaptive step sizing. Adaptive step sizing is a standard feature in many flowpipe tools including SpaceEx [7]. A range $[\alpha, \beta]$ wherein $0 < \alpha \leq \beta$ is specified by the user for the time step size. The purpose of adaptive step sizing is to choose a step size $\delta \in [\alpha, \beta]$ such that the Picard operator[2] on the current flowpipe yields a TM with remainder interval that is contained in I_e. Our approach starts from $\delta = \beta$ and as long as the Picard operator fails to yield a remainder inside I_e, it updates δ by $\delta' := \lambda\delta$ using a discount factor λ which is set to 0.5 in our implementation. If $\delta < \alpha$, a diagnostic message is printed asking the user to either (a) decrease the lower bound on the adaptive time step α, (b) enlarge the interval I_e or (c) increase the approximation order. Note that either (a) or (c) slows down the overall computation, but is ultimately unavoidable if the dynamics are hard to approximate.

[2] Given an ODE $\frac{d\boldsymbol{x}}{dt} = F(\boldsymbol{x}, t)$, the Picard operator on a function $g(\boldsymbol{x}_0, t)$ is given by $\mathbb{P}_F(g)(\boldsymbol{x}_0, t) = \boldsymbol{x}_0 + \int_0^t F(g(\boldsymbol{x}_0, s), s)\, ds$. If the Picard operator is contractive on g with some t, then g is an over-approximation of the flow at time t.

Adaptive TM orders. Adaptive choice of TM orders also seeks to bound the error within I_e. Our technique first concentrates on the state variables for which the interval error estimate is breached. The orders of these state variables are increased by 1. If the technique fails to achieve the interval I_e, the orders of all the remaining variables are increased as well. This process continues until the upper limit specified by the user is breached. On the other hand, if the interval I_e is achieved, our approach starts to decrease the TM orders to find the smallest order for which the flowpipe's remainder is contained in I_e. If adapting the TM orders fails, the tool prints a diagnostic.

Currently, the techniques of adaptive orders and step sizes operate independently of each other. The tool fixes the step sizes and performs adaptive TM orders; or fixes the TM orders and adapts the step sizes. The simultaneous adaptation of both step sizes and TM orders is not supported. This is also a challenging problem, since many optimal points of tradeoff may exist. Exploring these choices systematically in a time-efficient manner will be part of our future work.

FLOW* also provides various options for aggregating flowpipe/guard intersections. Users are allowed to partially or fully specify a parallelotopic template for the union.

3 Experimental Evaluation

We now provide an experimental evaluation of FLOW*. The evaluation includes a comparison against VNODE-LP [3] by Nedialkov et al. [11], a state-of-the art guaranteed integration tool for continuous systems. Next, we consider the evaluation for hybrid systems. Our previous work demonstrated a comparison against the Ariadne and the HySAT/iSAT tool. We have been unable to obtain tools from other papers to enable a meaningful comparison. Therefore, we restrict ourselves to showcasing performance on a new class of benchmarks and comparison of our new features against the earlier prototype. All experiments were performed on a i7-860 2.8GHz CPU with 4GB RAM running Ubuntu Linux. The benchmarks can be downloaded as a part of our release.

Continuous Systems: Table 1 shows a comparison with the VNODE-LP tool. The VNODE-LP tool often fails to integrate these benchmarks when the initial set is too large ("`VNODE - LP could not reach t = [T,T]`"). Therefore, to ensure a fair comparison, we subdivide the initial set into smaller intervals and integrate each separately. The G.S. column denotes the *grid sizes* used for the initial sets. Our approach subdivides the initial sets uniformly. We manually find the largest grid size for which VNODE-LP does not fail. This setting is chosen for the experiments. For experiments $\#5, 6, 7$, VNODE-LP failed or could not complete within the set time out of an hour. In contrast, FLOW* performs well on all the benchmark examples, often finishing well within the one hour timeout interval. The parameters used for FLOW* are provided for reference in Table 1.

The precision comparisons are quite tricky. FLOW* computes TM flowpipe segments whereas VNODE-LP computes boxes. Converting TMs into boxes can be quite expensive. Our approach uses a coarse computation using interval arithmetic. Nevertheless, the comparison in terms of the *max. widths* of the intervals (i.e, the maximum interval

[3] `http://www.cas.mcmaster.ca/~nedialk/Software/VNODE/VNODE.shtml`

Table 1. Comparisons on continuous benchmarks. All times are in seconds. Legends: var: number of the variables, T: time horizon is [0,T], orders: the adaption range of the TM orders, R.E.: remainder estimation, R.W.: remainder width, G.S.: grid size, W.S.: width of the solution interval for $\boldsymbol{x}(T)$, T.O.: > 1 hour.

			FLOW*							VNODE-LP		
ID	Benchmarks	var	T	step	orders	R.E.	time	R.W.	W.S.	G.S.	time	W.S.
1	Brusselator	2	10	0.02	3∼5	[-1e-4,1e-4]	8.2	<**1e-5**	**3e-2**	2e-2	**1.9**	3.3e-2
2	Lorentz	3	1	0.01	3∼6	[-1e-4,1e-4]	**15**	<**5e-4**	1.08	1e-2	64	**1.06**
3	Rössler	3	5	0.02	4∼6	[-5e-4,5e-4]	**30**	<**5e-4**	2.60	2e-2	50	**1.95**
4	6-D sys. [12]	6	1	0.01	3∼6	[-1e-2,1e-2]	**102**	<**1e-3**	**0.37**	2.5e-2	180	0.40
5	6-D sys. [12]	6	2	0.01	3∼6	[-1e-2,1e-2]	**213**	<**1e-3**	**0.37**	1.5e-2	T.O.	-
6	Reaction [13]	7	1e-3	2e-6	3∼5	[-1e-2,1e-2]	**469**	<**1e-1**	**15.91**	2e-2	Fail	-
7	Phosphorelay [14]	7	2	2e-3	3∼5	[-1e-3,1e-3]	**882**	<**1e-4**	**0.17**	1.5e-3	Fail	-
8	Phosphorelay [14]	7	3	2e-3	3∼5	[-1e-3,1e-3]	**836**	<**1e-6**	**3e-2**	1e-3	3156	3.5e-2
9	Bio [15]	9	0.1	1e-3	3∼5	[-1e-2,1e-2]	**121**	<**1e-1**	**1.42**	1e-2	318	1.58
10	Bio [15]	9	0.1	1e-3	3∼5	[-1e-2,1e-2]	**153**	<**1e-1**	**2.11**	1e-2	T.O.	-

width along any of the dimensions) is quite similar. Overall when successful, VNODE-LP's flowpipe was quite similar to that of FLOW*. We are investigating better box and octagon approximation schemes for TMs to enable a better comparison.

Hybrid Systems: We demonstrate our approach on a series of non-linear navigation benchmarks representing a vessel moving through a fluid. We assume a velocity dependent drag force along each direction $F_x : -k \cdot v_x^3$ and $F_y : -k \cdot v_y^3$ to the velocities v_x, v_y in each cell with $k = 0.1$. The other parameters are the almost the same as the linear benchmarks [16] with a few exceptions: the initial values for v_x in NAV05 lie in the range $[0.8, 0.1]$, and for NAV09, $x(0) \in [3.1, 3.5], v_y(0) \in [-0.8, -0.5]$. Table 2 summarizes the performance on a set of hybrid system benchmarks, comparing

Table 2. Hybrid benchmarks. Legends: var: # of variables, loc: # of locations, T: time horizon [0,T], jps: max jump depth, δ: step sizes, t: time cost (s), MUL: multiple TM orders. The safety properties are all proved.

					fixed steps&orders			adaptive steps			adaptive orders		
benchmarks	var	loc	T	jps	δ	order	t	δ	order	t	δ	order	t
n.-l. NAV04	4	7	30	8	0.01	3	138	[0.01,0.1]	3	57	0.05	3∼6	**32**
n.-l. NAV05	4	7	30	8	0.02	4	168	[0.01,0.1]	4	69	0.05	3∼6	**38**
n.-l. NAV06	4	7	30	8	0.01	3	162	[0.01,0.1]	3	70	0.05	3∼6	**40**
n.-l. NAV07	4	14	30	10	0.01	4	567	[0.01,0.1]	4	117	0.05	3∼6	**61**
n.-l. NAV08	4	14	30	10	0.01	4	545	[0.01,0.1]	4	122	0.05	3∼6	**60**
n.-l. NAV09	4	14	30	10	0.01	4	222	[0.01,0.02]	4	117	0.02	3∼6	**47**
Diabetic 1	4	9	360	6	0.02	4	1655	[0.01,0.1]	4	382	0.04	MUL	**212**
Diabetic 2	4	6	360	4	0.02	4	1023	[0.01,0.1]	4	229	0.04	MUL	**142**
Diabetic 3	5	9	360	6	0.02	4	852	[0.01,0.1]	4	191	0.04	MUL	**102**
Diabetic 4	5	6	360	4	0.02	4	502	[0.01,0.1]	4	106	0.04	MUL	**77**
Water tank	5	2	300	10	0.02	3	828	[0.01,0.1]	3	255	0.05	MUL	**218**

the effect of adaptive step sizes and adaptive orders. Interestingly, our results indicate that adapting the orders is more advantageous than step sizes. Adapting the TM orders seems to have a pronounced effect on the efficiency of the flowpipe guard intersection procedure. We also include the artificial pancreas (AP) models described in [4] with modified safety specifications. We created new benchmarks instances Diabetic 3 & 4 by adding timing delays between controller mode changes in Diabetic 1 & 2.

References

1. Berz, M.: Modern Map Methods in Particle Beam Physics. Advances in Imaging and Electron Physics, vol. 108. Academic Press (1999)
2. Berz, M., Makino, K.: Verified integration of ODEs and flows using differential algebraic methods on high-order Taylor models. Reliable Computing 4, 361–369 (1998)
3. Fainekos, G., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science 410, 4262–4291 (2009)
4. Chen, X., Ábrahám, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: Proc. RTSS 2012, pp. 183–192. IEEE (2012)
5. Makino, K., Berz, M.: Taylor models and other validated functional inclusion methods. J. Pure and Applied Mathematics 4(4), 379–456 (2003)
6. Neher, M., Jackson, K.R., Nedialkov, N.S.: On Taylor model based integration of ODEs. SIAM Journal on Numerical Analysis 45, 236–262 (2006)
7. Frehse, G., et al.: SpaceEx: Scalable verification of hybrid systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 379–395. Springer, Heidelberg (2011)
8. Platzer, A.: Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics. Springer (2010)
9. Benvenuti, L., Bresolin, D., Casagrande, A., Collins, P., Ferrari, A., Mazzi, E., Sangiovanni-Vincentelli, A., Villa, R.: Reachability computation for hybrid systems with Ariadne. In: Proc. the 17th IFAC World Congress. IFAC Papers-OnLine (2008)
10. Fränzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large non-linear arithmetic constraint systems with complex Boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)
11. Nedialkov, N.S.: Implementing a rigorous ode solver through literate programming. In: Modeling, Design, and Simulation of Systems with Uncertainties Mathematical Engineering, vol. 3, pp. 3–19. Springer (2011)
12. She, Z., Xue, B., Zheng, Z.: Algebraic analysis on asymptotic stability of continuous dynamical systems. In: Proc. ISSAC 2011, pp. 313–320. ACM (2011)
13. Craciun, G., Tang, Y., Feinberg, M.: Understanding bistability in complex enzyme-driven reaction networks. Proc. of the National Academy of Sciences 103(23), 8697–8702 (2006)
14. Klipp, E., Herwig, R., Kowald, A., Wierling, C., Lehrach, H.: Systems Biology in Practice: Concepts, Implementation and Application. Wiley-Blackwell (2005)
15. Vilar, J.M.G., Kueh, H.Y., Barkai, N., Leibler, S.: Mechanisms of noise-resistance in genetic oscillators. Proc. of the National Academy of Sciences 99(9), 5988–5992 (2002)
16. Fehnker, A., Ivančić, F.: Benchmarks for hybrid systems verification. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 326–341. Springer, Heidelberg (2004)

Efficient Robust Monitoring for STL

Alexandre Donzé[1], Thomas Ferrère[2], and Oded Maler[2]

[1] University of California, Berkeley, EECS Dept.
donze@eecs.berkeley.edu
[2] Verimag, CNRS and Grenoble University
{maler,ferrere}@imag.fr

Abstract. Monitoring transient behaviors of real-time systems plays an important role in model-based systems design. Signal Temporal Logic (STL) emerges as a convenient and powerful formalism for continuous and hybrid systems. This paper presents an efficient algorithm for computing the robustness degree in which a piecewise-continuous signal satisfies or violates an STL formula. The algorithm, by leveraging state-of-the-art streaming algorithms from Signal Processing, is linear in the size of the signal and its implementation in the Breach tool is shown to outperform alternative implementations.

1 Introduction

Temporal Logic (TL) is a popular formalism, introduced into systems design [Pnu77] as a language for specifying acceptable behaviors of reactive systems. Traditionally, it has been used for formal verification, either by deductive methods [MP91, MP95], or algorithmic methods (model checking [CGP99, QS82]). In this framework, the behaviors in question are typically discrete, that is, sequences of states and/or events. Two other assumptions concerning this use of TL in verification are implicit:

1. Systems correctness is affirmed if *all* system behaviors satisfy the specification. Thus model checking is based on composing the system model with an automaton for the specification and analyzing all possible paths in the combined transition system;
2. The satisfaction of a property by a behavior is a purely discrete matter (yes/no), which is in the spirit of most logics.[1]

In recent years, several trends suggest alternative ways to use TL in the *design* of complex systems and also during their *operations*. The first trend is due to the state-explosion wall, which limits the size of systems that can be verified (not to mention systems like programs with numerical variables or hybrid systems where verification is not even decidable). As a result we can see a proliferation of statistical methods a-la Monte-Carlo, where universal quantification is replaced

[1] There are some branches of multi-valued logic such as Fuzzy [Zad65] and probabilistic [Nil86] but mainstream Logic is about *true* and *false*.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 264–279, 2013.

by random simulation, with and even without statistical coverage guarantees. In this framework, also known as *runtime verification*, *assertion checking* or *monitoring*, the temporal formula is still used for a rigorous specification of the requirements, but unlike model-checking, it is evaluated on a *single* behavior at a time, a much easier task.

Unlike formal verification, monitoring does *not* require a *model* of the system. All it needs is a process that generates observable behaviors. As such it can be applied to systems which are viewed as *black boxes* either because their developers want to protect their intellectual property or because it is a complex program without a decent and tractable formal model. For the same reason, temporal property checking can be integrated in monitoring and diagnostics of *real* systems during their operation and provide more refined means to define and detect hazardous situations.

The present paper is based on *signal temporal logic* (STL), a formalism for specifying properties of dense-time real-valued signals [MN04, MNP08], for which a monitoring tool called AMT [NM07] has been developed and used in the context of analog and mixed-signal circuits [JKN10, MN12]. In many real-life applications, especially when dealing with continuous dynamics and numerical quantities, yes/no answers provide only partial information and could be augmented with *quantitative* information about the satisfaction to provide a better basis for decision making. To illustrate, consider the formula $x < c$ for constant c and a real-valued variable x ranging over some domain X. The formula splits X into $X^0 = \{x : x \geq c\}$ and $X^1 = \{x : x < c\}$. The latter is called the *validity domain* of the formula. When we pick a number $x \in X$, the answer to the satisfaction query $x \models \mathbf{x} < c$ depends on the *membership* of x in X^1 but not on its relative position inside or outside X^1. The *robustness degree* of the satisfaction should tell us whether x satisfies the formula by far ($x \ll c$) or very marginally ($x = c - \epsilon$ for a small positive ϵ). For this example, the robustness degree is captured by $c - x$ whose *sign* indicates satisfaction/violation and its *magnitude* indicates the *distance* between x and the *boundary* between X^0 and X^1.

Such notions have been introduced into TL by Fainekos and Pappas [FP09] for STL and by Fages and Rizk [RBFS08] for LTL over real-valued sequences. The robustness information is useful to assess the severity of a detected malfunctioning in a working system. It can also increase the confidence in the results of incomplete-coverage validation techniques, if it so happens that all sampled behaviors satisfy the requirements robustly. In a previous paper [DM10] we have introduced notions of robustness both in space and time, and provided an algorithm for computing the robustness degree with respect to a given signal. Signals are represented as sequences of time-stamped points and are interpreted as piecewise-linear via interpolation.

The major contribution of this paper is a new optimal algorithm that computes the robustness degree for such a signal in time linear with respect to the size of the signal (number of points). This algorithm guarantees that the overhead added by monitoring to the simulation process is acceptable, thus making

robustness-based monitoring a feasible technology that can be used routinely as an add-on for simulation engines. This low complexity is due to two key ideas:

- The use of the optimal streaming algorithm of Daniel Lemire [Lem06] to compute the min and max of a numeric sequence over a sliding window;
- The rewriting of the (bounded) timed "until" operator [DT04] as a conjunction of simpler timed and untimed operators.

The algorithm has been implemented at the core of Breach [Don10] which is a highly versatile toolbox for simulation-based analysis of complex systems, recently applied to biological reaction networks [DFG+11, MDMF12], to mine requirements of Simulink models in the automative industry [JDDS13] and to characterize patterns in musical signals [DMB+12]. Our implementation outperforms the tool S-TaLiRo [ALFS11], to the best of our knowledge the only other tool implementing quantitative semantics for dense time.

The rest of the paper is organized as follows. In Section 2, we recall the main definitions of STL and its quantitative semantics. In Section 4 we present our robustness computation framework and describe the algorithms for simple operators, such as negation and conjunction. Section 4 treats in details the case of untimed *until* and timed *eventually*, which completes the algorithms presentation. Section 5 discusses the theoretical worst-case complexity of the computation and Section 6 provides experimental results.

2 Signal Temporal Logic

In this section we recall the framework set in [MN04] to specify properties of real-valued signals, we extend it to a multi-valued logic as proposed by [FP09], and present the main properties of this extension.

We adopt the following conventions. The set of Boolean values is taken as $\mathbb{B} := \{\bot, \top\}$, with $\bot < \top$, $-\top = \bot$ and $-\bot = \top$, inducing the well known algebra. We write $\overline{\mathbb{R}} := \mathbb{R} \cup \mathbb{B}$ for the totally ordered set of real numbers with smallest element $\bot$ and greatest element $\top$.

A *signal* will be a function $D \to E$, with D an interval of $\mathbb{R}^+$ and $E \subset \overline{\mathbb{R}}$. Signals with $E = \mathbb{B}$ are called *Boolean* signals, whereas those where $E = \mathbb{R}$ are *real-valued* signals. An execution *trace* w is a set of real-valued signals $\{x_1^w, ..., x_k^w\}$ defined over some interval D of $\mathbb{R}^+$, which is called the *time domain* of w. Such a trace can be "booleanized" through a set of threshold predicates[2] of the form $x_i \geq 0$. Signal Temporal Logic is then a simple extension of Metric Temporal Logic where real-valued variables $(x_i)_{i\in\mathbb{N}}$ are transformed into Boolean values via these predicates. The syntax of STL will be taken as follows:

$$\varphi := \text{true} \mid x_i \geq 0 \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \, \mathbf{U}_I \, \varphi$$

Here x_i are variables, and I is a closed, non-singular interval of $\mathbb{R}^+$. This includes bounded intervals $[a, b]$ and unbounded intervals $[a, +\infty)$ for any $0 \leq a < b$.

[2] More expressive predicates could be added in the form of a preprocessing step, which we do not include explicitely in our framework.

Let w be a trace of time domain D. The formula φ is said to be *defined* over a time interval $\mathrm{dom}(\varphi, w)$ given by the following rules: $\mathrm{dom}(\mathrm{true}, w) = \mathrm{dom}(x_i \geq 0, w) = D$, $\mathrm{dom}(\neg\varphi, w) = \mathrm{dom}(\varphi, w)$, $\mathrm{dom}(\varphi \wedge \psi, w) = \mathrm{dom}(\varphi, w) \cap \mathrm{dom}(\psi, w)$, $\mathrm{dom}(\varphi \,\mathbf{U}_I\, \psi, w) = \{t \in \mathbb{R} \mid t, t+\inf(I) \in \mathrm{dom}(\varphi, w) \text{ and } t+\inf(I) \in \mathrm{dom}(\psi, w)\}$.

Boolean Semantics. For a trace w, the validity of an STL formula φ at a given time $t \in \mathrm{dom}(\varphi, w)$ is set according to the following inductive definition.

$$
\begin{array}{ll}
w, t \vDash \mathrm{true} & \\
w, t \vDash x_i \geq 0 & \text{iff } \; x_i^w(t) \geq 0 \\
w, t \vDash \neg\varphi & \text{iff } \; w, t \nvDash \varphi \\
w, t \vDash \varphi \wedge \psi & \text{iff } \; w, t \vDash \varphi \text{ and } w, t \vDash \psi \\
w, t \vDash \varphi \,\mathbf{U}_I\, \psi & \text{iff } \; \text{exists } t' \in t + I \text{ s.t. } w, t' \vDash \psi \text{ and for all } t'' \in [t, t'] \,, w, t'' \vDash \varphi
\end{array}
$$

We can redefine other usual operators as syntactic abbreviations:

$$
\begin{array}{ll}
\mathrm{false} := \neg\mathrm{true} & \varphi \vee \psi := \neg(\neg\varphi \wedge \neg\psi) \\
\Diamond_I\, \varphi := \mathrm{true}\, \mathbf{U}_I\, \varphi & \Box_I\, \varphi := \neg\Diamond_I\, \neg\varphi
\end{array}
$$

We use $\Diamond$ and $\mathbf{U}$ as shorthands for *untimed* operators $\Diamond_{[0,+\infty)}$ and $\mathbf{U}_{[0,+\infty)}$. For a given formula φ and execution trace w, we define the *satisfaction signal* $\chi(\varphi, w, .)$ as follows:

$$
\text{for all } t \in \mathrm{dom}(\varphi, w), \quad \chi(\varphi, w, t) := \begin{cases} \top \text{ if } w, t \vDash \varphi \\ \bot \text{ otherwise} \end{cases}
$$

Monitoring the satisfaction of a formula φ can be done by computing for each subformula ψ of φ the entire satisfaction signal $\chi(\psi, w, .)$. The procedure is recursive on the structure of the formula, and goes from the atomic predicates up to the top formula [MN04].

Quantitative Semantics. Given a formula φ, trace w, and time $t \in \mathrm{dom}(\varphi, w)$, we define the quantitative semantics $\rho(\varphi, w, t)$ by induction as follows:

$$
\begin{aligned}
\rho(\mathrm{true}, w, t) &= \top \\
\rho(x_i \geq 0, w, t) &= x_i^w(t) \\
\rho(\neg\varphi, w, t) &= -\rho(\varphi, w, t) \\
\rho(\varphi \wedge \psi, w, t) &= \min\{\rho(\varphi, w, t), \rho(\psi, w, t)\} \\
\rho(\varphi \,\mathbf{U}_I\, \psi, w, t) &= \sup_{t' \in t+I} \min\{\rho(\psi, w, t'), \inf_{t'' \in [t, t']} \rho(\varphi, w, t'')\}
\end{aligned}
$$

It is worth noting that if we let $\chi(x_i \geq 0, w, t) = \begin{cases} \top \text{ if } x_i^w(t) \geq 0 \\ \bot \text{ otherwise} \end{cases}$ and apply to χ the above inductive rules, we fall back to Boolean signals and obtain an equivalent characterization of χ. In the quantitative semantics however, atomic predicates $x_i \geq 0$ do not evaluate to $\top$ or $\bot$ but give a real value representing the

distance to satisfaction or to violation, which is then propagated in the formula using the $\{\min, \max, -\}$ operations on $\overline{\mathbb{R}}$.

From the lattice properties of $(\overline{\mathbb{R}}, <)$, we are granted the axioms of associativity, commutativity, neutral element, and distributivity. The minus function remains involutive, which gives us the usual de Morgan laws $\neg(\varphi \vee \psi) \sim \neg\varphi \wedge \neg\psi$ and $\neg\Diamond_I \varphi \sim \Box_I \neg\varphi$. Derived operators enjoy the same natural interpretation as in the Boolean semantics: $\rho(\varphi \vee \psi, w, t) = \max\{\rho(\varphi, w, t), \rho(\psi, w, t)\}$, $\rho(\Diamond_I \varphi, w, t) = \sup_{t' \in t+I} \rho(\varphi, w, t')$, and $\rho(\Box_I \varphi, w, t) = \inf_{t' \in t+I} \rho(\varphi, w, t')$.

Property of Robustness Estimate. The quantitative semantics of STL have two fundamental properties, that would alone justify their introduction. Firstly, whenever $\rho(\varphi, w, t) \neq 0$ its sign indicates the satisfaction status.

Theorem 1 (Soundness). *Let φ be an STL formula, w a trace and t a time.*

$$\rho(\varphi, w, t) > 0 \Rightarrow w, t \vDash \varphi$$
$$\rho(\varphi, w, t) < 0 \Rightarrow w, t \nvDash \varphi$$

Secondly, if w satisfies φ at time t, any other trace w' whose pointwise distance from w is smaller than $\rho(\varphi, w, t)$ also satisfies φ at time t.

Theorem 2 (Correctness). *Let φ be an STL formula, w and w' traces over the same time domain, and $t \in \mathrm{dom}(\varphi, w)$.*

$$w, t \vDash \varphi \text{ and } \|w - w'\|_\infty < \rho(\varphi, w, t) \quad \Rightarrow \quad w', t \vDash \varphi$$

On these grounds we now talk of ρ as the *robustness estimate*. For a given trace w, and φ an STL formula, we will refer to the *robustness signal* of φ with respect to w, as the signal $\rho(\varphi, w, .)$. Similarly to the satisfaction signal, it is defined over the time domain $\mathrm{dom}(\varphi, w)$.

Until Rewrite. The properties $\varphi \,\mathbf{U}_{[a,b]}\, \psi \sim \Diamond_{[a,b]} \psi \wedge \varphi \,\mathbf{U}_{[a,+\infty)}\, \psi$ and $\varphi \,\mathbf{U}_{[a,+\infty)}\, \psi \sim \Box_{[0,a]} (\varphi \,\mathbf{U}\, \psi)$ extend from Boolean to quantitative semantics.

Lemma 1. *For two STL formula φ, ψ, a trace w and any time t where defined,*

$$\rho(\varphi \,\mathbf{U}_{[a,b]}\, \psi, w, t) = \rho(\Diamond_{[a,b]} \psi \wedge \varphi \,\mathbf{U}_{[a,+\infty)}\, \psi, w, t) \quad (1)$$
$$\rho(\varphi \,\mathbf{U}_{[a,+\infty)}\, \psi, w, t) = \rho(\Box_{[0,a]} (\varphi \,\mathbf{U}\, \psi), w, t) \quad (2)$$

Proof. We only prove the first rewrite rule, the second can be obtained by a similar argument. We note y, y' the robustness signals of φ, ψ relative to w. Let $u := \sup_{\tau \in t+[a,b]} \min\{y'(\tau), \inf_{[t,\tau]} y\}$ and $v := \min\left\{ \sup_{t+[a,b]} y', \sup_{\tau \geq t+a} \min\{y'(\tau), \inf_{[t,\tau]} y\}\right\}$ be the robustness values of equation (1) for some given time t. Suppose that $u \neq v$, for instance $u < v$. We define the signals $x : t \mapsto y(t) - \frac{u+v}{2}$ and $x' : t \mapsto y'(t) - \frac{u+v}{2}$. Now consider the formulas $\gamma := (x \geq 0) \,\mathbf{U}_{[a,b]}\, (x' \geq 0)$ and $\theta := \Diamond_{[a,b]} (x' \geq 0) \wedge (x \geq 0) \,\mathbf{U}_{[a,+\infty)}\, (x' \geq 0)$. Pushing the constant $\frac{u+v}{2}$ outside min, sup and inf in their quantiative semantics we get $\rho(\gamma, w, t) = u - \frac{u+v}{2} < 0$ and $\rho(\theta, w, t) = v - \frac{u+v}{2} > 0$, so that $w, t \nvDash \gamma$ while $w, t \vDash \theta$ by Theorem 1. This is clearly impossible, as γ and θ are equivalent in the Boolean semantics.

3 Computing the Robustness Estimate

While monitoring a system, real or simulated, signals are available to us as finite timed words over the alphabet $\mathbb{R}^n$. We interpret these by linear interpolation. This section presents the basic framework for computing robustness under this hypothesis, using the following high-level procedure.

Algorithm 1. Robustness(φ, w)

```
switch (φ)
case true:
   return ⊤̄   % a constant ⊤ signal
case x_i ≥ 0:
   return x_i^w
case ∗ φ_1:
   y := Robustness(φ_1, w)
   return Compute(∗, y)
case φ_1 ∗ φ_2:
   y := Robustness(φ_1, w)
   y' := Robustness(φ_2, w)
   return Compute(∗, y, y')
end switch
```

Definition 1. *A signal y is said to be finitely piecewise-linear, continuous (*f.p.l.c. *for short) if there exists a finite sequence $(t_i)_{i \leq n_y}$ such that:*

- *the definition domain of y is $[t_0, t_{n_y})$*
- *for all $i < n_y$, y is continuous at t_i and affine on $[t_i, t_{i+1})$*

$(t_i)_{i \leq n_y}$ will be called the time sequence *of y.*

Let us note $dy(t)$ the derivative of y at time t. In what follows, any signal in the observed trace will be assumed to be f.p.l.c. Such a signal will be represented by its sequence $(t_i, y(t_i), dy(t_i))_{i<n_y}$, along with cut-off time t_{n_y}. As we assume continuity, this representation is slightly redundant, but facilitates the splitting of signals into segments. We may in addition require the limit of y at t_{n_y}, for which we abuse the notation and simply write $y(t_{n_y})$.

We will see that for every operator, the quantitative semantics preserves the f.p.l.c. property of signals, so that we can always assume such signals as inputs of the calculation in the inductive step. Note that continuity is clearly preserved by the sup and inf operations. Also, no new derivative value is created in the process, so that from a computational standpoint the overhead of handling derivatives is compensated by the interpolation speedup.

Boolean Operators. Computing the robustness signal of $\neg\varphi$ from that of φ is trivial. One can simply note that if the sequence $(t_i, y(t_i), dy(t_i))_{i<n_y}$ represents

$\rho(\varphi, w, .)$ then the sequence $(t_i, -y(t_i), -dy(t_i))_{i<n_y}$ represents $\rho(\neg\varphi, w, .)$. For conjunction, let us take y and y' the robustness signals of φ and ψ respectively, producing z the robustness signal of $\varphi \wedge \psi$. We build the sequence $(r_i)_{i\leq n_z}$ containing the sampling points of y and y' when they are both defined, and the points where y and y' punctually intersect. Note that there are less than $n_y + n_{y'}$ such intersections and at most $n_y + n_{y'}$ sampling points, so that we have $n_z \leq 4 \cdot \max\{n_y, n_{y'}\}$. Now, for all $i < n_z$ we let, using the lexicographic order

$$\begin{pmatrix} z(r_i) \\ dz(r_i) \end{pmatrix}^t = \min\left\{ \begin{pmatrix} y(r_i) \\ dy(r_i) \end{pmatrix}^t, \begin{pmatrix} y'(r_i) \\ dy'(r_i) \end{pmatrix}^t \right\}$$

The resulting sequence $(r_i, z(r_i), dz(r_i))_{i<n_z}$ adequately represents $\rho(\varphi \wedge \psi, w, .)$.

Untimed Eventually $\Diamond$. Although not primitive in the syntax, this operator is easily computed and will be used as a subroutine for the until computation. We take y the robustness signal of φ, $(t_i)_{i<n_y}$ its time sequence, and z the robustness signal z of $\Diamond\,\varphi$. For any t in its definition domain $z(t) = \sup_{t'\geq t} y(t')$, and we have immediately the following property: $\forall s < t,\ z(s) = \max\{\sup_{[s,t)} y, z(t)\}$.
The step computation can be derived by applying the property at $t = t_{i+1}$ the time of some sample $i + 1 < n_y$. Depending on the possible orderings of $\{y(t_i), y(t_{i+1}), z(t_{i+1})\}$ there are four possibilities,

$$\begin{aligned}
y(t_i) \leq y(t_{i+1}):&\quad \forall s \in [t_i, t_{i+1}),\ z(s) = \max\{y(t_{i+1}), z(t_{i+1})\} \\
y(t_i) > y(t_{i+1}) \geq z(t_{i+1}):&\quad \forall s \in [t_i, t_{i+1}),\ z(s) = y(s) \\
z(t_{i+1}) \geq y(t_i) > y(t_{i+1}):&\quad \forall s \in [t_i, t_{i+1}),\ z(s) = z(t_{i+1}) \\
y(t_i) > z(t_{i+1}) > y(t_{i+1}):&\quad \exists t^*\, \forall s \in [t_i, t^*),\ z(s) = y(s) \text{ and} \\
&\qquad\qquad \forall s \in [t^*, t_{i+1}),\ z(s) = z(t_{i+1})
\end{aligned}$$

The induction is initialized by substituting $\perp$ for $z(t_{n_y})$ in the property. Over the whole signal y, each sample t_i of y generates up to two samples in z, so that $n_z \leq 2 \cdot n_y$.

Until. We treat the case of timed until by rewriting it into untimed until and timed eventually. This decomposition, due to [DT04] has been successfully applied to monitoring STL for the Boolean semantics. We have proved (Lemma 1) that it also holds for the quantitative semantics. For unbounded until $\mathbf{U}_{[a,+\infty)}$ we can use directly rewrite rule (2), whereas for bounded until we have $\varphi\,\mathbf{U}_{[a,b]}\,\psi \sim \Diamond_{[a,b]}\,\psi \wedge \Box_{[0,a]}\,(\varphi\,\mathbf{U}\,\psi)$ by rules (1) and (2). *Globally* being the dual of *eventually*, it only remains to develop an algorithm for $\Diamond_{[a,b]}$ with $[a, b]$ a non-singular interval, along with an algorithm for the untimed until. These more involved computations are the object of the next section.

4 Algorithms

From a close examination of operators $\neg$, $\wedge$ and $\Diamond$ just achieved, we can immediately derive the corresponding Compute() algorithms with time-complexity linear in the number of samples of their input signals. By duality we also have an algorithm for $\vee$ with the same property. We now give detailed algorithms for the remaining two operators: $\mathbf{U}$, and $\Diamond_{[a,b]}$.

For any signal y and two time instants $s < t \in \mathbb{R}^+$ we note $y_{\restriction[s,t)}$ the restriction of y to the time interval $[s,t)$. The output signal z will be computed as a series of segments $z_{\restriction[s,t)}$ for s,t extracted from the time sequence of the input signals.

Operator $\mathbf{U}$. Let y and y' be the robustness signals of φ and ψ respectively, with $(t_i)_{i\leq n_y}$ and $(t'_i)_{i\leq n_{y'}}$ their respective time sequences. The calculation outputs z, the robustness signal of $\varphi\,\mathbf{U}\,\psi$ relative to w. By definition we have $z(t) = \sup_{\tau\in[t,+\infty)} \min\{y'(\tau), \inf_{[t,\tau]} y\}$.

Similarly to the Boolean semantics [MN04], the computation can be done by backward induction. Let $s < t$ be two times in $\mathrm{dom}(\varphi\,\mathbf{U}\,\psi)$. If we define $z_t(s) := \sup_{\tau\in[s,t)} \min\{y'(\tau), \inf_{[s,\tau]} y\}$, by the general properties of sup and inf we obtain the following inductive formula:

$$z(s) = \max\left\{z_t(s), \min\{\inf_{[s,t)} y, z(t)\}\right\}$$

Suppose that y is affine on the interval $[s,t]$.

- If $dy(s) \leq 0$ then $\forall\tau \in [s,t)$, $\inf_{[s,\tau]} y = y(\tau)$. Thus $z_t(s) = \sup_{\tau\in[s,t)} \min\{y'(\tau), y(\tau)\}$, and $z(s) = \max\left\{z_t(s), \min\{y(t), z(t)\}\right\}$
- Otherwise $\forall\tau \in [s,t)$, $\inf_{[s,\tau]} y = y(s)$. Therefore $z_t(s) = \sup_{\tau\in[s,t)} \min\{y'(\tau), y(s)\} = \min\{y(s), \sup_{[s,t)} y'\}$, and $z(s) = \max\left\{z_t(s), \min\{y(s), z(t)\}\right\}$.

We let $t = t_i$ in the above, and compute $z(s)$ for all s on the segment $[t_i, t_{i+1})$. Taking the notation $\overline{v}$ for the constant signal of value v, we can now express all the operations involved as computations previously implemented. This gives us Algorithm 2, written under the simplifying assumption that $[t_0, t_{n_y}) \subseteq \mathrm{dom}(\psi, w)$, which can always be achieved by interpolation of y at $t'_0, t'_{n_{y'}}$ and sample renumbering if necessary.

Lemma 2. *The time-complexity of Algorithm 2 is linear in* $\max\{n_y, n_{y'}\}$.

Proof. The algorithm takes n_y steps. Step i computes the signal z on the interval $[t_i, t_{i+1})$ from partial signals $y_{\restriction[t_i,t_{i+1})}$ and $y'_{\restriction[t_i,t_{i+1})}$ along with two constant signals. Each step uses algorithms linear in the size of their inputs, so that the execution takes time linear in the sum of the size of the inputs of each step which is at most $3 \cdot n_y + n_{y'}$. Thus the total execution time is linear in $\max\{n_y, n_{y'}\}$.

Algorithm 2. Compute($\mathbf{U}, y, y'$)

$z_0 := \overline{\bot}$
$i := n_y - 1$
while $i \geq 0$ **do**
  if $dy(t_i) \leq 0$ **then**
    $z_1 := \text{Compute}(\Diamond, y'_{\restriction [t_i, t_{i+1})})$
    $z_2 := \text{Compute}(\wedge, z_1, y_{\restriction [t_i, t_{i+1})})$
    $z_3 := \text{Compute}(\wedge, \overline{y(t_{i+1})}, z_0)$
    $z_{\restriction [t_i, t_{i+1})} := \text{Compute}(\vee, z_2, z_3)$
  else
    $z_1 := \text{Compute}(\wedge, y'_{\restriction [t_i, t_{i+1})}, y_{\restriction [t_i, t_{i+1})})$
    $z_2 := \text{Compute}(\Diamond, z_1)$
    $z_3 := \text{Compute}(\wedge, y_{\restriction [t_i, t_{i+1})}, z_0)$
    $z_{\restriction [t_i, t_{i+1})} := \text{Compute}(\vee, z_2, z_3)$
  end if
  $i := i - 1$
  $z_0 := \overline{z(t_{i+1})}$
end while
return z

Operator $\Diamond_{[a,b]}$. Let y be the robustness signal of φ with respect to w, with $(t_i)_{i \leq n_y}$ its time sequence. For a given $I = [a, b]$ we want to compute the robustness signal of $\Diamond_I \varphi$ with respect to w, defined as $z : t \mapsto \sup_{t+I} y$.

Let t be a given time instant in $\text{dom}(\Diamond_I \varphi, w)$. Due to the f.p.l.c. hypothesis on y, one can easily see that there exists $t^* \in t + I$ such that $y(t^*) = z(t)$. Moreover it is sufficient to consider candidates for the maximum in the time sequence of y, along with $t + a$ and $t + b$. Namely

$$\sup_{t+[a,b]} y = \max \{y(t+a), y(t+b)\} \cup \{y(t_i) \mid t_i \in t + (a, b]\}$$

The problem of computing z is thus reduced to computing the maximum of $\{y(t_i) \mid t_i \in t + (a, b]\}$ when non empty, followed by a pointwise maximum with $y(t+a), y(t+b)$. Intuitively, time intervals where $\{y(t_i) \mid t_i \in t+(a, b]\}$ provide the maximum corresponding to "plateau" phases, where the supremum is reached at a point in the interior of the interval $t + I$. On the other hand, intervals where $y(t + a)$ or $y(t + b)$ give the maximum correspond to descending and ascending phases, respectively.

The maximum of $\{y(t_i) \mid t_i \in t + (a, b]\}$ can be computed by a straightforward adaptation of the running maximum filter algorithm given by [Lem06]. This work addresses the problem of computing, for signals over time domain $\mathbb{N}$, the maximum over a shifting window consisting of k elements. It it the first algorithm with time complexity linear in the length of the signal and *independent* of the window size k. We generalize this algorithm to the case of variable time-step.

The main idea is to maintain, as we increase t a set of indices M, so-called a monotonic edge, such that

$$i \in M \text{ iff } t_i \in t + I \text{ and for all } t_j > t_i \text{ in } t + I,\ y(t_j) < y(t_i)$$

In particular for any given t, if $M \neq \emptyset$ we have $y(t_{\min M}) = \max\{y(t_i) \mid t_i \in t + (a, b]\}$. Assume M is known for a given time s, and is non empty. We begin by finding the first $t > s$ so that a either a new point appears at $t + b$ or some maximum candidate in M disappears at $t+a$, or both. We update M accordingly: if $t_{\min M} = t + a$ we remove $\min M$, the first index from M. Then if $t + b = t_i$ for a given i then we compare $y(t_i)$ with $y(t_k)$ for $k \in M$ in decreasing order of k, starting with the last candidate $y(t_{\max M})$. If $k \in M$ is so that $t_k \leq t_i$ then t_k is removed from M as outperformed by t_i, otherwise we stop. At this point i is inserted as the new last element of M. We now have in M an ordered set of maximum candidates in $t+I$; we can output $y(t_{\min M})$ and repeat the procedure for the next event. The algorithm steps are illustrated in Figure 1.

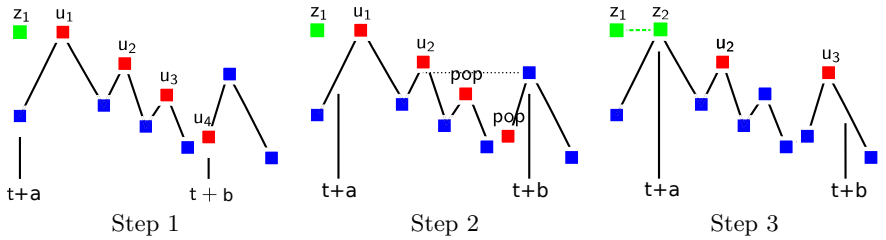


Fig. 1. Steps of the Lemire algorithm. Initially, M contains indexes u_1, u_2, u_3, u_4. At step 2, a new value appears at $t + b$ which removes u_3 and u_4. At step 3, $t + a$ reaches u_1 which is removed from M.

We obtain the full algorithm, for piecewise linear signals, by integrating this value with $y(t+a)$ and $y(t+b)$. Let us note $y_a : t \mapsto y(t+a)$ and $y_b : t \mapsto y(t+b)$; such signals can be computed by simple shift of the time sequence of y. We see $y(t_{\min M})$ as a constant signal, noted $\overline{y(t_{\min M})}$. We then use the $\vee$ algorithm to take the pointwise maximum of y_a, y_b and $\overline{y(t_{\min M})}$. Note that on a step interval $[s, t)$ of our computation, these three signals are always affine. The pseudo-code of Algorithm 3 details the overall operation. For simplicity's sake we ignore the case whereby $M = \emptyset$ (occurs if $[a, b]$ is finer than some time step). In such a case we would only need to place $i + 1$ in M and output the pointwise maximum of y_a, y_b on the current segment.

Lemma 3. *The time-complexity of Algorithm 3 is linear in n_y.*

Proof. We begin by noticing that the computation of signals y_a, y_b and y' use previous algorithms linear in n_y. For each step, the integration of y' with the constant signal $\overline{y(t_{\min M})}$ over the whole domain involves at most 3 samples;

Algorithm 3. Compute($\Diamond_{[a,b]}, y$)

$y_a := \text{Shift}(y, -a)$, $y_b := \text{Shift}(y, -b)$, $y' := \text{Compute}(\vee, y_a, y_b)$
$s := t_0 - b,\ t := s,\ i := 0,\ M := \{0\}$
while $t + a < t_{n_y}$ **do**
  $t := \min\{t_{\min M} - a, t_{i+1} - b\}$
  if $t = t_{\min M} - a$ **then**
    $M := M \setminus \{\min M\}$
    $s := t$
  end if
  if $t = t_{i+1} - b$ **then**
    while $y(t_{i+1} - b) \geq y(t_{\max M})$ and $M \neq \emptyset$ **do**
      $M := M \setminus \{\max M\}$
    end while
    $M := M \cup \{i + 1\}$
    $i := i + 1$
  end if
  if $s \geq t_0$ **then**
    $z_{\restriction [s,t)} := \text{Compute}(\vee, y'_{\restriction [s,t)}, \overline{y(t_{\min M})})$
  end if
end while
return z

there are at most $2 \cdot n_y$ steps, so that the overall cost of these operations is linear in n_y. All that is left to show is that maintaining M throughout the computation takes time linear in n_y.

Storing M as a doubly-linked queue, we keep a sorted array and the elements to be accessed are always at the front or the back. Therefore we can consider the cost of each operation on M as unitary, and independent of the size of M. Under this hypothesis the cost of computing M is proportional to the number of value comparisons involved. With the same argument as [Lem06] we notice that on the whole run, there are n_y elements entering M and thus there are also n_y elements are leaving M. Each time the comparison $y(t_{i+1} - b) \geq y(t_{\max M})$ evaluates to true, an element is removed from M so there can only be n_y such comparisons that evaluate to true. When it evaluates to false we leave the loop hence there are at most n_y such comparisons that evaluate to false. Over the whole execution Algorithm 3 uses at most $2 \cdot n_y$ such value comparisons, making its execution time linear in n_y.

5 Complexity

To keep the discussion short, we restrict the syntax to primitive connectors $\{\text{true}, x_i \geq 0, \neg, \wedge, \mathbf{U}_I\}$. The complexity results of previous sections can be summed up by stating the following: there exists a constant A such that for any signals y, y' and any connector $\neg, \wedge$ or $\mathbf{U}_I$, the corresponding Compute() algorithm takes execution time smaller than $A \cdot \max\{n_y, n_{y'}\}$.

We are interested in the time complexity of the robustness computation with respect to both trace and formula size. A trace $w = \{x_1, x_2, ..., x_k\}$ will have for size $|w| := \max\{n_{x_1}, n_{x_2}, ...n_{x_k}\}$, the maximum number of samples of its signals. A formula φ is represented by its parse tree, in which each node is an STL operator. A path in the tree has a *length* taken to be the number of binary connectors $\wedge$ and $\mathbf{U}_I$ it contains. The height of the formula $\mathrm{h}(\varphi)$ is then defined as maximum length for paths in the tree of φ. Note that our definition of height ignores atoms and negations. The size of a formula $|\varphi|$ will be simply defined as the number of nodes in the tree of φ.

Theorem 3. *There exists a constant d such that for any φ, w, the signal $z := \rho(\varphi, w, .)$ has a number of samples $n_z \leq d^{\mathrm{h}(\varphi)} \cdot |w|$*

Proof. If we take y, y' two signals and z be one of $t \mapsto \min\{y(t), y'(t)\}$ or $t \mapsto \sup_{\tau \in t+[a,b]} \min\{y'(\tau), \inf_{[t,\tau]} y\}$, then by immediate consequence of the existence of linear time algorithms to compute such signals there exists d such that $n_z \leq d \cdot \max\{n_y, n_{y'}\}$. We now prove the property by induction on the structure of φ.

For atomic formulas, the height is 0 while the robustness signal has a number of sampling points at most the size of the input trace. For the negation, we have $h(\neg\varphi) = h(\varphi)$ by definition while the number of samples is unchanged, we conclude by the induction hypothesis. Finally we examine the induction step for conjunction, the case of timed until is similar.

By definition we have $\mathrm{h}(\varphi \wedge \psi) = \max\{\mathrm{h}(\varphi), \mathrm{h}(\psi)\} + 1$. Let y, y' and z be the robustness signals of φ, ψ and $\varphi \wedge \psi$ with respect to w. By the inductive hypothesis we have $n_y \leq d^{\mathrm{h}(\varphi)} \cdot |w|$, and $n_{y'} \leq d^{\mathrm{h}(\psi)} \cdot |w|$. From the introductory remark $n_z \leq d \cdot \max\{d^{\mathrm{h}(\varphi)} \cdot |w|, d^{\mathrm{h}(\psi)} \cdot |w|\} = d \cdot d^{\max\{\mathrm{h}(\varphi), \mathrm{h}(\psi)\}} \cdot |w| = d^{\mathrm{h}(\varphi \wedge \psi)} \cdot |w|$.

Corollary 1. *The algorithm Robustness(φ, w) has time-complexity in $\mathcal{O}(|\varphi| \cdot d^{\mathrm{h}(\varphi)} \cdot |w|)$.*

Proof. Let φ be an arbitrary formula, and w an arbitrary trace. By Theorem 3, each subformula ψ of φ has a robustness signal with at most $d^{\mathrm{h}(\psi)} \cdot |w|$ sampling points, which is smaller or equal to $d^{\mathrm{h}(\varphi)} \cdot |w|$ in particular. Thus for each node of the tree of φ the corresponding Compute() algorithm takes execution time at most $A \cdot d^{\mathrm{h}(\varphi)} \cdot |w|$. There are exactly $|\varphi|$ such nodes, so that the main robustness computation of φ with respect to w is achieved in time at most $A \cdot |\varphi| \cdot d^{\mathrm{h}(\varphi)} \cdot |w|$.

The next example will convince the reader that the robustness signal size can indeed increase exponentially with the height of the formula.

Example 1. Let w_0 be the trace with signal x, defined over $[0, 8)$ and represented by the sequence $\big(t_i, x(t_i), dx(t_i)\big)_{i<4} := \{(0,0,0), (5,0,-1), (6,-1,0), (7,-1,1)\}$. We define by induction the formula sequence $(\varphi_k)_{k \in \mathbb{N}}$ by

$$\varphi_0 := x \geq 0 \qquad \varphi_{k+1} := \Box_{[0, \frac{1}{2^{k+1}}]} \left(\Diamond_{[0, \frac{1}{2^{k-1}}]} \varphi_k \wedge \Diamond_{[\frac{1}{2^{k-1}}, \frac{1}{2^{k-2}}]} \varphi_k\right)$$

One can easily show (see Figure 2) that n_k the number of samples of $\rho(\varphi_k, w_0, .)$ is $2^k \cdot |w_0|$, while $\mathrm{h}(\varphi_k) = 3 \cdot k$, so that we have $n_k = (\sqrt[3]{2})^{\mathrm{h}(\varphi_k)} \cdot |w_0|$. Note that

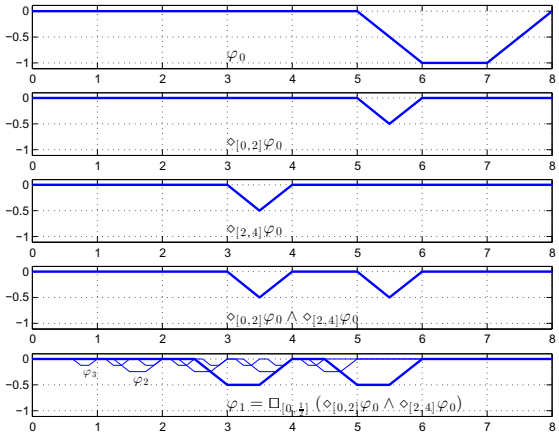


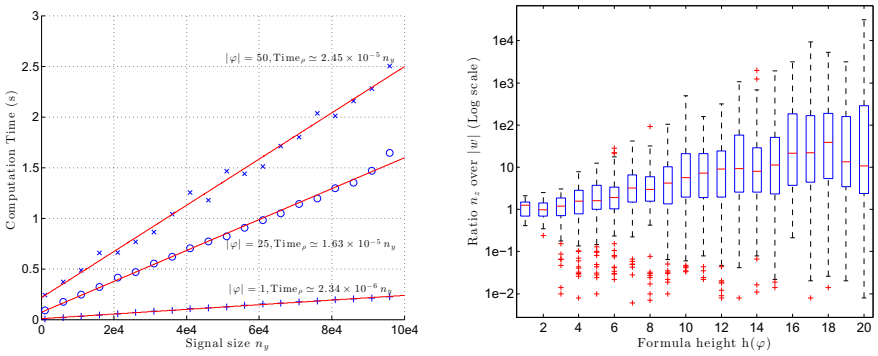
Fig. 2. Some steps in the computation of $\rho(\varphi_k, w_0, .)$

this example entails the same behaviour for the size of the *satisfaction* signals with respect to the formula height, so that traditional Boolean monitoring also appears to suffer some level of exponential complexity to this respect.

6 Experiments

The proposed algorithm has been implemented in C++ and interfaced with the STL parser of Breach. We conducted experiments using signals formed by i.i.d samples with uniform distribution. To plot Figure 3-(a), we computed the robustness estimate for three formulas of size 1, 25 and 50 for signals of size ranging from 10^4 to 10^5 samples. We could confirm that for each formula, the computation time is linear with respect to the signal size. For Figure 3-(b), we fixed $n_y = 1000$ and generated 20 times 100 formulas of height $\mathrm{h}(\varphi)$ ranging from 1 to 20 and as many signals of size n_y. Our goal was to explore experimentally the result of Theorem 3. For each pair of formula and signal, we computed the robust signal and the ratio $\alpha = \frac{n_z}{n_y}$ between its samples size n_z and the input signal size $|w| = n_y$. We first observe that for each value of $h(\varphi)$, there is high variability of this ratio. For instance, for $\mathrm{h}(\varphi) = 20$, α can vary from 10^{-2} to more than 10^4. Secondly, we observe that as predicted by Theorem 3, α appears to be bounded by some exponential $d^{\mathrm{h}(\varphi)}$, where we can roughly estimate $d \simeq 1.7$. Finally, the average value of α on our experiment seems also to follow an exponential function of $\mathrm{h}(\alpha)$, though with a lower constant $d \simeq 1.12$. This seems to indicate that exponential complexity is rather the rule than the exception. However, the conditions of these experiments (random signals, formula maximizing heights) are likely to be much more chaotic than for real systems and specifications. In particular, other experiments (not reported here)

suggest that "high" formulas resulting mostly from large numbers of conjunctions do not exhibit an exponential behavior, so that the complexity mostly arise from nested temporal operators. And we believe that human-written specifications are not likely to contain deeply nested temporal operators, as the intuition of such formula is generally hard to grasp.



(a) Compution time against signal size

(b) Size increase of robust signal against h(φ).

Fig. 3. Experimental validation of the complexity results given in Section 5

Next we compared the performance of our algorithm with that of the DP-TaLiRo [FSUY12], based on a dynamic programming approach and implemented in S-TaLiRo version 1.3. We compared the monitoring of $\Diamond_I$ and $\mathbf{U}_I$ for signals of various sizes and different time intervals I.[3] The results are given in Table 1. Except for signals of small size (100-1000), Breach is consistently faster by several orders of magnitude. One partial explanation could be the fact that in the framework of TaLiRo, the robust satisfaction of a predicate is obtained through the computation of a distance function, which is done by the monitoring algorithm, leading to an additional "hidden cost" [FSUY12]), whereas in our case, this computation is separated from the monitoring.[4] However, all the predicates in our experiments are such that the distance function should be trivial to compute so this alone cannot account for the difference in the performance. Also, our results confirmed that the computation time for bounded time operators does not depend on the size of the time interval, as in [Lem06], which is an improvement over the complexity of the DP-TaLiRo algorithm.

[3] Note that a comparison of nested formulas would make less sense since DP-TaLiRo is implemented for fixed time-steps, thus the number of samples cannot vary from one level to another.

[4] In Breach, predicates are of the general form $f(x) > 0$ and here we considered that f was computed beforehand to produce the signals $y(t) = f(x(t))$ as inputs to our algorithm. Note that this is slightly more general than TaLiRo as f can implement a distance function [JDDS13]).

Table 1. Experimental comparison with DP-Taliro algorithm

Computation times (s) of robustness estimates for $\Diamond_I$ and $\mathbf{U}_I$								
	DP-Taliro				Breach			
n_y	$1e2$	$1e3$	$1e4$	$1e5$	$1e2$	$1e3$	$1e4$	$1e5$
$\Diamond_{[1,2]}$	0.001091	0.00278	0.176	16.6	0.00312	0.00302	0.004	0.0193
$\Diamond_{[1,11]}$	0.000689	0.00304	0.212	20.4	0.00286	0.00262	0.00391	0.0173
$\Diamond_{[1,21]}$	0.000713	0.00334	0.253	24.3	0.00268	0.00269	0.00412	0.0185
$\Diamond_{[1,31]}$	0.000707	0.0038	0.278	27.3	0.00302	0.00281	0.00409	0.0208
$\mathbf{U}_{[1,2]}$	0.523	4.72	46.8	486	0.00577	0.00766	0.0268	0.228
$\mathbf{U}_{[1,11]}$	0.482	4.55	47.1	493	0.00567	0.00743	0.0269	0.223
$\mathbf{U}_{[1,21]}$	0.468	4.59	46.2	499	0.00545	0.00722	0.0268	0.229
$\mathbf{U}_{[1,31]}$	0.462	4.7	46.7	505	0.00567	0.0073	0.0274	0.222

7 Discussion and Future Work

We developed a new algorithm computing robust satisfaction of STL formulas by piecewise-linear signals. The algorithm is linear in the size of the signal as measured by the number of sampling points. The algorithm extends to dense time the algorithm of [Lem06] for maintaining the maximal value of a numerical sequence over a shifting window and its implementation confirms its theoretical properties. Our implementation could handle signals with millions of samples and formulas with tens of operators. It remains to be seen whether the worst-case exponential growth in the size of the formula occurs for real-life formulas rather than the random formulas we experimented with. Another important direction to investigate would be the design of an online variant of our algorithm, which is by nature offline.

Acknowledgments. We thank Dejan Nickovic and anonymous referees for useful comments. This work was supported by the French ANR projects Syne2Arti and EQINOCS and by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

[ALFS11] Annpureddy, Y., Liu, C., Fainekos, G., Sankaranarayanan, S.: S-taLiRo: A tool for temporal logic falsification for hybrid systems. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 254–257. Springer, Heidelberg (2011)

[CGP99] Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (1999)

[DFG+11] Donzé, A., Fanchon, E., Gattepaille, L.M., Maler, O., Tracqui, P.: Robustness analysis and behavior discrimination in enzymatic reaction networks. PLoS One 6(9) (2011)

[DM10] Donzé, A., Maler, O.: Robust satisfaction of temporal logic over real-valued signals. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 92–106. Springer, Heidelberg (2010)

[DMB+12] Donzé, A., Maler, O., Bartocci, E., Nickovic, D., Grosu, R., Smolka, S.: On temporal logic and signal processing. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 92–106. Springer, Heidelberg (2012)

[Don10] Donzé, A.: Breach, A toolbox for verification and parameter synthesis of hybrid systems. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 167–170. Springer, Heidelberg (2010)

[DT04] D'Souza, D., Tabareau, N.: On timed automata with input-determined guards. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 68–83. Springer, Heidelberg (2004)

[FP09] Fainekos, G.E., Pappas, G.J.: Robustness of temporal logic specifications for continuous-time signals. Theoretical Computer Science 410(42) (2009)

[FSUY12] Fainekos, G.E., Sankaranarayanan, S., Ueda, K., Yazarel, H.: Verification of automotive control applications using s-taliro. In: ACC (2012)

[JDDS13] Jin, X., Donzé, A., Deshmukh, J., Seshia, S.: Mining requirements from closed-loop control models. In: HSCC 2013 (2013)

[JKN10] Jones, K.D., Konrad, V., Nickovic, D.: Analog property checkers: a DDR2 case study. Formal Methods in System Design 36(2) (2010)

[Lem06] Lemire, D.: Streaming maximum-minimum filter using no more than three comparisons per element. CoRR, abs/cs/0610046 (2006)

[MDMF12] Mobilia, N., Donzé, A., Moulis, J.-M., Fanchon, E.: A model of the cellular iron homeostasis network using semi-formal methods for parameter space exploration. In: HSB (2012)

[MN04] Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)

[MN12] Maler, O., Nickovic, D.: Monitoring properties of analog and mixed-signal circuits. Software Tools for Technology Transfer (2012)

[MNP08] Maler, O., Nickovic, D., Pnueli, A.: Checking temporal properties of discrete, timed and continuous behaviors. In: Avron, A., Dershowitz, N., Rabinovich, A. (eds.) Pillars of Computer Science. LNCS, vol. 4800, pp. 475–505. Springer, Heidelberg (2008)

[MP91] Manna, Z., Pnueli, A.: The Temporal Logic of Reactive and Concurrent Systems: Specification. Springer, New York (1991)

[MP95] Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety. Springer, New York (1995)

[Nil86] Nilsson, N.J.: Probabilistic logic. Artificial intelligence 28(1), 71–87 (1986)

[NM07] Nickovic, D., Maler, O.: AMT: A property-based monitoring tool for analog systems. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 304–319. Springer, Heidelberg (2007)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: Proc. 18th Annual Symposium on Foundations of Computer Science (FOCS), pp. 46–57 (1977)

[QS82] Queille, J.P., Sifakis, J.: Specification and Verification of Concurrent Systems in CESAR. In: 5th Int. Symp. on Programming (1982)

[RBFS08] Rizk, A., Batt, G., Fages, F., Soliman, S.: On a continuous degree of satisfaction of temporal logic formulae with applications to systems biology. In: Heiner, M., Uhrmacher, A.M. (eds.) CMSB 2008. LNCS (LNBI), vol. 5307, pp. 251–268. Springer, Heidelberg (2008)

[Zad65] Zadeh, L.: Fuzzy sets. Information and Control 8, 338–353 (1965)

Abstraction Based Model-Checking of Stability of Hybrid Systems

Pavithra Prabhakar and Miriam Garcia Soto

IMDEA Software Institute

Abstract. In this paper, we present a novel abstraction technique and a model-checking algorithm for verifying Lyapunov and asymptotic stability of a class of hybrid systems called *piecewise constant derivatives*. We propose a new abstract data structure, namely, *finite weighted graphs*, and a modification of the predicate abstraction based on the *faces* in the system description. The weights on the edges trace the distance of the executions from the origin, and are computed by using linear programming. Model-checking consists of analyzing the finite weighted graph for the absence of certain kinds of cycles which can be solved by dynamic programming. We show that the abstraction is sound in that a positive result on the analysis of the graph implies that the original system is stable. Finally, we present our experiments with a prototype implementation of the abstraction and verification procedures which demonstrate the feasibility of the approach.

1 Introduction

With the ubiquitous use of embedded control systems, particularly in safety critical application areas such as avionics and automotive, there is an ever increasing need for scalable automated verification of embedded programs. A unique feature of these programs is their interaction with a physical world which is typically continuous, and hence the behavior of the system as a whole consists of mixed discrete-continuous components. The verification of hybrid systems has remained a stubbornly challenging problem due to the complex behavior exhibited by these systems. It is a well known fact that the problem of verifying even simple properties such as safety is undecidable for a class of hybrid systems with relatively simple dynamics [13]. Hence, much effort has been invested in investigating approximate analysis methods, which can be broadly grouped into the following categories.

One direction involves methods for *approximate post analysis*. Computing the post of a set of states is a crucial step in state-space exploration based model-checking algorithms. The exact post computation is feasible for only a small subclass of hybrid systems; and hence the focus has been on computing approximations of the post sets. Various efficient methods and representations for the post set have been proposed including zonotopes, polytopes, ellipsoids and support functions [10,11,24]. Bounded error approximations of the entire hybrid

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 280–295, 2013.

systems have also been explored, and include methods such as hybridization and polynomial approximations [27,2,21,5].

The other set of results concern *abstraction based analysis*, where a simplified model of the system is computed and analysed to infer correctness properties about the original model. Various methods for computing abstract models include predicate abstraction, hybrid abstractions [8,22,1,6]. These methods in general do not provide any bound on the error of the abstraction; hence, they are usually accompanied by an abstraction-refinement loop based on, for example, counter-example analysis. Most of the above techniques can be classified as model-checking algorithms and mainly focus on safety verification. There have also been deductive verification techniques for safety verification [26,19].

In this paper, we focus on a different class of properties, namely, stability. Stability properties capture the notion that small perturbations to the initial state or input to the system result in only small changes to the future behaviors of the system. Stability is an important property expected out of every control system so much so that a system which is not stable is deemed useless. The study of stability analysis of purely continuous systems is a mature field and there exist characterizations and automatically verifiable methods for their analysis [15]. Most of these methods can be classified under deductive verification, where the proof of stability is established by exhibiting a Lyapunov function [18,17], which represents an "energy" function which decreases over the trajectories of the system. There also exist extensions of Lyapunov function based analysis methods for hybrid systems [16,7,4,12], which either exhibit a common Lyapunov function for the entire hybrid system or combine the analysis of a Lyapunov function for each mode of the hybrid system. However, constructing Lyapunov functions in an extremely arduous task; and often requires the ingenuity of the designer. Hence, the state-of-the-art technique for proving stability of hybrid systems relies on automating the search for Lyapunov functions. Complete results in this direction only exist for certain subclasses of systems such as purely continuous linear dynamical systems, where it is known that a quadratic function always exists and can be found by solving linear matrix inequalities.

In contrast to the traditional approach to stability analysis, the focus of this paper is in exploring model-checking as an alternate method for stability analysis. The standard methods of abstraction such as predicate abstraction [1,6] do not suffice for this purpose, since as pointed out in [23], the notion of simulation does not suffice to preserve stability and instead needs a stronger notion of simulation strengthened with continuity requirements.

The main contribution of this paper is an abstraction procedure and a model-checking algorithm for verifying stability of a class of hybrid systems called *piecewise constant derivatives* studied in [3]. Our abstraction procedure can be interpreted as a modified predicate abstraction procedure. The abstract model is a finite weighted graph whose nodes correspond to the faces of the regions defined by the predicates. The edges correspond to the existence of an execution from one face to the other, similar to the predicate abstraction; however, in addition, it is also annotated with a weight which measures the closeness of the

execution to the origin between its end-points. The finite weighted graph is then analysed for the absence of cycles in which the product of the weights on the edges exceeds 1, and the absence of such cycles implies stability of the system. We consider two classical notions of stability, namely, Lyapunov and asymptotic stability. Our analysis procedure is conservative in general, however, for the two dimensional case, it coincides with the decidability algorithm in [25]. The paper [25] explores the decidability boundary for Lyapunov and asymptotic stability and shows the decidability in 2 dimensions for a class of hybrid systems slightly more general than *PCD*s and undecidability for *PCD*s with 5 dimensions.

There has been some work on predicate abstraction based analysis for certain weaker notions of stability, such as, region stability, which require that all trajectories of the system reach a region in a finite time [20,9].

Though we focus on a simple class of systems, we believe that the technique of face based abstraction into finite weighted graph and its analysis can be carried over to more complex classes of hybrid systems. Below we summarize the main features and contributions of the paper.

1. A face based abstraction procedure to reduce the verification of Lyapunov and asymptotic stability in a sound manner to the analysis on a finite weighted graph.
2. A model-checking algorithm for analyzing the abstract finite weighted graph.
3. The model-checking algorithm returns an "abstract counter-example" which is crucial in developing an abstraction refinement framework.
4. A prototype implementation of the abstraction and verification procedure, which demonstrates the feasibility of the technique.

In contrast to the previous works on deductive verification based analysis for Lyapunov and asymptotic stability, our method has the flavor of an algorithmic verification technique based on state-space exploration. Additional details can be found at the following url:
http://software.imdea.org/people/pavithra.prabhakar/Details/home.html

2 Overview of the Abstraction and Model-Checking Algorithm

In this section, we will illustrate with examples the main ideas behind the abstraction based model-checking algorithm.

A piecewise constant derivative system (PCD) [3] is a hybrid system which partitions the state-space into some finite number of polyhedral regions and associates a flow vector with each region specifying the direction in which the state evolves while in the region. For example, shown in Figure 2 is a two-dimensional PCD which has four regions corresponding to the four quadrants, r_1 with flow $(-1, 1)$, r_2 with flow $(-1, -1)$, r_3 with flow $(1, -1)$ and r_4 with flow $(1, 1)$. A sample run of the PCD is the trajectory σ which starts as $(1, 0)$ and moves along a straight line to $(0, 1)$, then to $(-1, 0)$, $(0, -1)$ and back to $(1, 0)$.

This is a finite execution of the system. Repeating σ an infinite number of times gives a complete execution, an execution in which the time diverges to infinity.

Intuitively, Lyapunov stability captures the notion that every execution starting close to the origin remains close to the origin. More precisely, it says that given any $\epsilon > 0$, we can find a $\delta > 0$, such that all trajectories starting within distance δ from the origin remain with in distance ϵ of the origin at all times. Note that starting at any point in any δ-unit square around the origin, the executions of the system remain with in a $\sqrt{2}\delta$ unit square around the origin. Hence, the system is stable. On the other hand, if the flow associated with r_4 were $(2, 1)$ instead of $(1, 1)$, then for every δ, there exist trajectories which start within distance δ but diverge from the origin.

The other canonical notion of stability is *asymptotic stability* which requires along with Lyapunov stability that all complete executions converge to the origin. The 2 dimensional system in Figure 2 is not asymptotically stable because the complete execution corresponding to repeating the execution σ, which moves along the rhombus, infinitely many times does not converge to the origin.

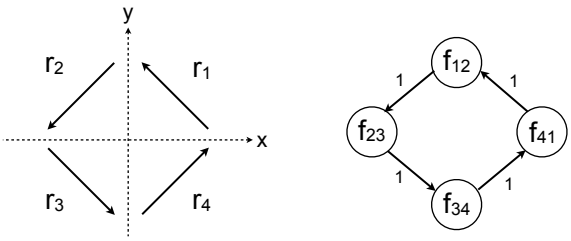


Fig. 1. 2 dimensional system and its abstraction

The main idea behind the analysis is to construct a finite weighted graph which contains information about the distance from origin of the executions of the system. More precisely, the nodes of the graph correspond to the faces of the regions. And there is an edge between two faces if there exists an execution which starts at some point on the first face and reaches some point on the other face while remaining in the common region. This part is similar in some sense to predicate abstraction except that we consider the faces instead of the regions. The additional element is the weights on the edges of the graph. The weight corresponds to the maximum scaling in the distance to the origin between any point in the first face and the points reachable from it on the second face. Abstract graph for the 2 dimensional example in shown on the right in Figure 2. It consists of four faces f_{12}, f_{23}, f_{34} and f_{41}, where f_{ij} corresponds to the common set of points between region i and region j. The weight on all the edges is 1. For example, starting at distance δ on f_{41}, that is, $(\delta, 0)$, the unique point on f_{12} reachable is $(0, \delta)$. Hence the ratio of the distance of $(0, \delta)$ from the origin to the distance of $(\delta, 0)$ from the origin is $\delta/\delta = 1$.

The abstract graph has the property that corresponding to every execution of the original system, there is a path in the graph where each edge corresponds to the part of the execution which remains within a particular region. More over, the weight on the edge is an upper bound on the weight of the scaling associated with executions corresponding to the edge. The main theorem of the paper states that if the abstract graph does not contain a cycle such that the product of the weights is > 1 and the flow associated with all the regions is such that there exist no trajectories which diverge while remaining within the region, then the system is Lyapunov stable. Further, it is asymptotically stable if there are no cycles such that the product of the weights is 1. The theorem essentially states that the abstraction is sound with respect to stability analysis. It turns out that the theorem is in fact a necessary and sufficient condition for the 2 dimensional case as shown in [25]. However, the same is not true in higher dimensions as shown below.

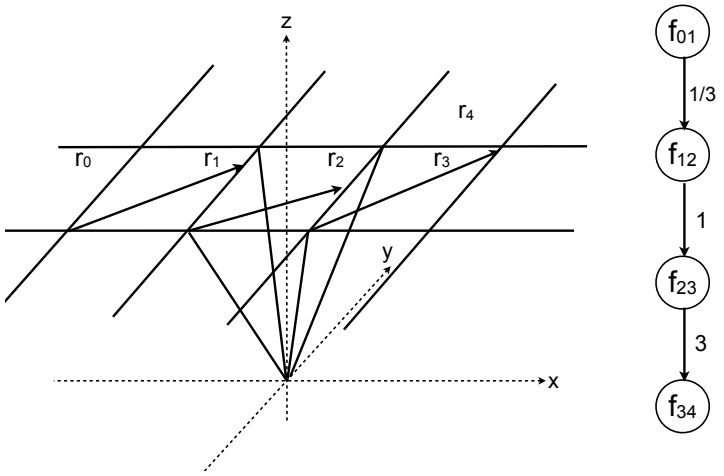


Fig. 2. 3 dimensional system and its abstraction

In the Figure 2, r_2 is the region given by $-z \leq x \leq z, -z \leq y \leq z, z \geq 0$. Similarly, r_0, r_1, r_3 are obtained by replacing the constraint on x by $x \leq -3z$, $-3z \leq x \leq z$ and $z \leq x \leq 3z$, respectively. r_4 is given by $z \leq x \leq 3z, z \leq y, z \geq 0$. The flow vectors of r_1, r_2 and r_3 are given by $(2, 1.5, 0), (2, 1, 0)$ and $(2, 2, 0)$, respectively. A part of the abstract graph corresponding to the faces f_{01}, f_{12}, f_{23} and f_{34} are shown in Figure 2. We use the infinity norm for the distance, that is, distance between two points (x_1, y_1, z_1) and (x_2, y_2, z_3) is given by $max\{|x_1 - x_2|, |y_1 - y_2|, |z_1 - z_3|\}$. The ratios on the edges are computed as before. For example consider the edge from f_{01} to f_{12}. Since the flow vectors are parallel to the xy-plane, it suffices to consider any particular value of z to compute the ratios. Let us fix $z = 1$. All the points on f_{01} are at distance 3. The points reached on f_{12} from points on f_{01} are at distance at most 1. In fact,

the point $(-1, 0.75, 1)$ at distance 1 from the origin is reachable from the point $(-3, 0.25, 1)$ on f_{01}, hence the ratio is $1/3$.

Note that there does not exist an execution corresponding to the path $f_{01}f_{12}f_{23}$, that is, a point on f_{01} such that there exists an execution from it which reaches f_{12} and continues to reach f_{23}. Though there exists an execution corresponding to the path $f_{12}f_{23}f_{34}$, there does not exist one in which the part of the execution corresponding to $f_{12}f_{23}$ has a scaling of 1 and the part corresponding to $f_{23}f_{34}$ has value 3. Hence, in general, the abstract graph is conservative.

3 Preliminaries

Notations. Let $\mathbb{R}$, $\mathbb{R}_{\geq 0}$, $\mathbb{Q}$ and $\mathbb{N}$ denote the set of reals, non-negative reals, rationals and natural numbers, respectively. We use $[n]$ to denote the set of natural numbers $\{1, \cdots, n\}$. Given a function F, we use $Dom(F)$ to denote the domain of F. Given a function $F : A \to B$ and a set $A' \subseteq A$, $F(A')$ will denote the set $\{F(a) \mid a \in A'\}$. Given a finite set A, $|A|$, denotes the number of elements of A.

Euclidean space. Let X denote the n-dimensional Euclidean space $\mathbb{R}^n$, for some n. Given $\mathbf{x} = (x_1, \cdots, x_n) \in X$ and $1 \leq i \leq n$, we use $\langle \mathbf{x} \rangle_i$ to denote the i-th component of x, namely, x_i. Given $\mathbf{x}, \mathbf{y} \in X$, we use the standard notation $\mathbf{x} \cdot \mathbf{y}$ to denote the dot product of the vectors $\mathbf{x}$ and $\mathbf{y}$ and $|\mathbf{x}|$ to denote the norm of $\mathbf{x}$. Most of our results are independent of the particular norm used, however, in some parts we will focus on the infinity norm, which is defined as $|\mathbf{x}| = max_{1 \leq i \leq n} \langle \mathbf{x} \rangle_i$. We use $B_\epsilon(\mathbf{x})$ to denote an open ball of radius ϵ around $\mathbf{x}$, that is, the set $\{\mathbf{y} \mid |\mathbf{x} - \mathbf{y}| < \epsilon\}$.

Sequences. Let *SeqDom* denote the set of all subsets of $\mathbb{N}$ which are prefix closed, where a set $S \subseteq \mathbb{N}$ is prefix closed if for every $m, n \in \mathbb{N}$ such that $n \in S$ and $m < n$, $m \in S$. A *sequence* over a set A is a mapping from an element of *SeqDom* to A. Given a sequence $\pi : S \to A$, we also denote the sequence by enumerating its elements in the order, that is, $\pi(0), \pi(1), \cdots$.

Linear Constraints and Convex Polyhedral Sets. A *linear expression* is an expression of the form $\mathbf{a} \cdot \mathbf{x} + \mathbf{b}$, where $\mathbf{x}$ is a vector of n variables and $\mathbf{a}, \mathbf{b} \in X$. A *linear constraint* or *predicate* is a term $e \sim 0$, where e is a linear expression and $\sim$ is a relational operator in $\{<, \leq, =\}$. A linear constraint $\mathbf{a} \cdot \mathbf{x} + \mathbf{b}$ is called *homogenous* if $\mathbf{b} = \mathbf{0}$. Given a linear constraint C given by $\mathbf{a} \cdot \mathbf{x} + b \sim \mathbf{0}$, we use $[\![C]\!]$ to denote the set of all values $\mathbf{v} \in \mathbb{R}^n$ such that $\mathbf{a} \cdot \mathbf{v} + \mathbf{b} \sim \mathbf{0}$. Given a set of linear constraints $\mathcal{C}$, we use $[\![\mathcal{C}]\!]$ to denote the convex set defined by $\mathcal{C}$, namely, $\bigcap_{C \in \mathcal{C}} [\![C]\!]$.

A *half-space* in X is a set which can be expressed as the set of all points $\mathbf{x} \in X$ satisfying a linear constraint, $\mathbf{a} \cdot \mathbf{x} + \mathbf{b} \sim \mathbf{0}$, where $\sim \in \{<, \leq\}$. A *convex polyhedral set* is an intersection of finitely many half-spaces. We will use $ConvPolyhed(X)$ to denote the set of all convex polyhedral subsets of X.

Given a convex polyhedral set P defined by the constraints $\{e_1 \sim_1 \mathbf{0}, \cdots, e_k \sim_k \mathbf{0}\}$, a *face* of P is a non-empty polyhedral set defined by constraints of the form $\{e_1 \sim'_1 \mathbf{0}, \cdots, e_k \sim'_k \mathbf{0}\}$, where each $\sim'_i$ is either $\sim_i$ or $=$, and at least one of the $\sim'_i$ is $=$. We denote the set of faces of a convex polyhedral set P by *Faces*(P).

Given a convex polyhedral set P defined by the constraints $\{e_1 \sim_1 \mathbf{0}, \cdots, e_k \sim_k \mathbf{0}\}$, the closure of P, denoted *Closure*(P), is the set defined by the constraints $\{e_1 \leq \mathbf{0}, \cdots, e_k \leq \mathbf{0}\}$. Essentially, every $e_i < 0$ is replaced by $e_i \leq 0$. Note that *Closure*(P) contains all the limit points of P.

Given a convex polyhedral set S and a point $\mathbf{s} \in S$, we call the *cone* of S at $\mathbf{s}$, the set of all vectors $\mathbf{v} \neq \mathbf{0}$ such that there exists a $t > 0$, for which $\mathbf{s} + \mathbf{v}t \in S$. We denote this set by *Cone*($S, \mathbf{s}$).

A *partition* $\mathcal{P}$ of $\mathbb{R}^n$ into convex polyhedral sets is a finite set of convex polyhedral sets $\{P_1, \cdots, P_k\}$ such that $\cup_{i=1}^{k} P_i = \mathbb{R}^n$ and for each $i \neq j$, $P_i \cap P_j = \emptyset$. We will call the P_is as regions.

Graphs. A *graph* G is a pair (V, E), where V is a finite set of nodes and $E \subseteq V \times V$ is a finite set of edges. A *path* π of a graph $G = (V, E)$ is a finite sequence of nodes $v_0, \cdots, v_k$ such that $(v_i, v_{i+1}) \in E$ for $0 \leq i < k$. The length of a path π, denoted $|\pi|$, is the number of edges occurring in it. A path π is *simple* if all the nodes occurring in the path are distinct. A *cycle* in a graph $G = (V, E)$ is a path $\pi = v_0, \cdots, v_k$ such that the first and the last nodes are the same, that is, $v_0 = v_k$. A cycle is *simple* if all the nodes except the last one are distinct.

A node v is *reachable* from a node u if there exists a path whose first element is u and the last element is v, that is, there exists a π such that $\pi(0) = u$ and $\pi(|Dom(\pi)| - 1) = v$.

We associate weights with the edges of a graph using weighting functions. A *weighted graph* is a triple $G = (V, E, W)$, where (V, E) is a graph and $W : E \to \mathbb{R}_{\geq 0} \cup \{\infty\}$ is a weighting function. We extend the weight function to a path as follows. Weight of a path π of G, denoted $W(\pi)$, is $\Pi_{0 \leq i < |Dom(\pi)|-1} w((\pi(i), \pi(i+1)))$.

4 Piecewise Constant Derivative System (*PCD*)

In this section, we introduce a formal model for a subclass of hybrid systems called piecewise constant derivative systems [3]. Hybrid automata [14] are a popular formal model for systems with mixed discrete-continuous behaviors. A hybrid automaton combines the finite automaton model for discrete systems and the differential equation formalism for continuous dynamics. Here we consider a class of systems in which the statespace is partitioned into a finite number of regions with each region corresponding to a discrete mode; and the continuous dynamics is provided by a differential equation of the form $\dot{\mathbf{x}} = \mathbf{a}$, where $\mathbf{x}$ is a vector of variables and $\mathbf{a}$ is a constant vector. Hence, in each region the variables evolve at a constant rate given by the vector associated with the region and is allowed to switch to a neighboring region at the common boundary. Next, we define the system formally.

Definition 1. *An n-dimensional PCD $\mathcal{H} = (\mathcal{P}, \varphi)$, where $\mathcal{P}$ is a partition of $\mathbb{R}^n$ into convex polyhedral sets and $\varphi : \mathcal{P} \to \mathbb{R}^n$ is the flow function.*

The semantics of a *PCD* $\mathcal{H}$ is given by the set of executions exhibited by the system. An execution of a *PCD* starting from a point $\mathbf{x} \in \mathbb{R}^n$ follows the flow associated with the region in which $\mathbf{x}$ lies until it reaches the boundary of a neighboring region upon which it switches to the flow of the new region.

We need some definitions. An *execution interval* is either $[0, T]$ representing the set $\{t \in \mathbb{R} \mid 0 \leq t \leq T\}$, $[0, T)$ representing the set $\{t \in \mathbb{R} \mid 0 \leq t < T\}$ or $[0, \infty)$ representing the set $\{t \in \mathbb{R} \mid 0 \leq t\}$. We denote the set of execution intervals by *IntExec*. Given an execution interval $I \in$ *IntExec*, we define its *size*, denoted *Size*(I), to be T if $I = [0, T]$ or $[0, T)$ and ∞ otherwise.

Definition 2. *An* execution σ *of* $\mathcal{H} = (\mathcal{P}, \varphi)$ *is a function* $\sigma : I \to \mathbb{R}^n$, *where* $I \in$ *IntExec, such that there exists a finite or infinite sequence* $\eta = (P_1, \delta_1)(P_2, \delta_2) \cdots$ *satisfying for every* $1 \leq i \leq Dom(\eta)$*:*

- $P_i \in \mathcal{P}$ *and if* $i < Dom(\eta)$, $P_i \neq P_{i+1}$,
- δ_i *in* $\mathbb{R}_{\geq 0}/0$ *for* $i < Dom(\eta)$ *and if* $Dom(\eta)$ *is finite, then* $\delta_{|Dom(\eta)|}$ *in* $(\mathbb{R}_{\geq 0}/0) \cup \{\infty\}$.
- $I = [0, t_{Dom(\eta)}]$ *if* $Dom(\eta)$ *is finite and* $I = [0, t_{Dom(\eta)})$ *otherwise, where* $t_0 = 0$ *and for* $1 \leq j \leq Dom(\eta)$, $\sum_{l=1}^{j} \delta_l = t_j$, *and*
- *for every* $t \in I$, $\sigma(t) = \sigma(t_i) + \varphi(P_i)(t - t_i)$, *where* i *is the smallest integer such that* $t_i \leq t \leq t_{i+1}$.

We denote the set of all executions of $\mathcal{H}$ by *Exec*$(\mathcal{H})$. We call an execution σ with domain $[0, T]$ for some $T \in \mathbb{R}_{\geq 0}$ a *finite* execution and one with domain $[0, \infty)$ a *complete* execution.

Definition 3. *Given a finite execution* σ*, we define the* scaling *of* σ*, denoted Scaling*(σ)*, to be* $|\sigma(Size(Dom(\sigma)))|/|\sigma(0)|$.

Representation of a PCD. We represent the *PCD* by specifying a common set of linear predicates such that each of the regions is the conjunction of the predicates in either the positive or the negative form. More precisely, an n-dimensional *PCD* is represented by $\mathcal{H} = (\mathcal{C}, F)$, where $\mathcal{C}$ is a set of linear predicates and $F : 2^{\mathcal{C}} \to \mathbb{R}^n$ is the flow function. The regions of $\mathcal{H}$, denoted *Regions*$(\mathcal{C})$ or *Regions*$(\mathcal{H})$, are the non-empty sets $R \subseteq \mathbb{R}^n$ such that $R = \cap_{C \in A} [\![C]\!] \bigcap \cap_{C \notin A} \mathbb{R}^n \backslash [\![C]\!]$ for some $A \subseteq \mathcal{C}$. We call R the region *generated by* A. $\mathcal{H}$ consists of at most $2^{|\mathcal{C}|}$ polyhedral sets. The flow associated with a region R generated by $A \subseteq \mathcal{C}$ is $F(A)$. From now on, we use the both the notations for representing a *PCD*, and the particular notation will be clear from the context.

Notation. We will assume that the constants in the linear predicates and flows are all rational numbers.

5 Stability: Lyapunov and Asymptotic

In this section, we define two classical notions of stability in control theory, and state some general results about stability of *PCD*. We consider stability of the system with respect to an equilibrium point, which in our setting will be the origin.

Definition 4. $\mathbf{0}$ *is an* equilibrium point *of a PCD* $\mathcal{H}$ *if every execution of* $\mathcal{H}$ *starting at* $\mathbf{0}$ *remains at* $\mathbf{0}$*, that is, every execution* $\sigma \in Exec(\mathcal{H})$ *with* $\sigma(0) = \mathbf{0}$ *satisfies* $\sigma(t) = \mathbf{0}$ *for every* $t \in Dom(\sigma)$*.*

Intuitively, Lyapunov stability captures the notion that an execution starting close to the equilibrium point remains close to it, and asymptotic stability, in addition, requires that executions starting in a small neighborhood around the equilibrium point converge to it.

Definition 5. *A PCD* $\mathcal{H}$ *is said to be* Lyapunov stable*, if for every* $\epsilon > 0$*, there exists a* $\delta > 0$ *such that for every execution* $\sigma \in Exec(\mathcal{H})$ *with* $\sigma(0) \in B_\delta(\mathbf{0})$*,* $\sigma(t) \in B_\epsilon(\mathbf{0})$ *for every* $t \in Dom(\sigma)$*.*

We use $Lyap(\mathcal{H}, \epsilon, \delta)$ to denote the fact that for every execution $\sigma \in Exec(\mathcal{H})$ with $\sigma(0) \in B_\delta(\mathbf{0})$, $\sigma(t) \in B_\epsilon(\mathbf{0})$ for every $t \in Dom(\sigma)$. In fact, we do not need to consider all possible values for ϵ in the definition of Lyapunov stability but only values in a small neighborhood around $\mathbf{0}$.

Proposition 1. *A PCD* $\mathcal{H}$ *is Lyapunov stable if and only if there exists an* $\epsilon' > 0$ *such that for every* $0 < \epsilon < \epsilon'$*, there exists a* $\delta > 0$ *for which* $Lyap(\mathcal{H}, \epsilon, \delta)$ *holds.*

Definition 6. *A PCD* $\mathcal{H}$ *is said to be* asymptotically stable*, if it is Lyapunov stable and there exists a* $\delta > 0$ *such that every complete execution* $\sigma \in Exec(\mathcal{H})$ *with* $\sigma(0) \in B_\delta(\mathbf{0})$ *converges to* $\mathbf{0}$*, that is, for every* $\epsilon > 0$*, there exists a* $T \in Dom(\sigma)$*, such that* $\sigma(t) \in B_\epsilon(\mathbf{0})$ *for every* $t \geq T$*.*

We use $Asymp(\mathcal{H}, \delta)$ to denote the fact that every complete execution $\sigma \in Exec(\mathcal{H})$ with $\sigma(0) \in B_\delta(\mathbf{0})$ converges to $\mathbf{0}$.

6 Abstraction Based Model-Checking

In this section, we describe the abstraction based model-checking procedure for Lyapunov and asymptotic stability. The main steps in the procedure are as follows:

1. *Preprocessing*: We convert a *PCD* to a normal form in which all the constraints are homogenous.
2. *Abstraction*: We construct an abstract finite weighted graph using face based abstraction which is a sound abstraction for analyzing Lyapunov and asymptotic stability. The construction of the graph involves solving satisfiability of a set of linear constraints and linear programming problems.

3. *Model-Checking*: This involves analyzing the abstract graph for the absence of cycles with weight greater than 1 (or greater than or equal to 1) and analyzing the regions of the *PCD* for "non-explosion".

Next, we explain each of the steps in detail.

6.1 Normal Form for *PCD*

In this section, we present a reduction of a general *PCD* to one in a normal form. We say that a *PCD* is in normal form if each of the predicates defining the regions is homogeneous.

Definition 7. *A PCD* $\mathcal{H} = (\mathcal{C}, F)$ *is said to be in* normal form *if each of the predicates in* $\mathcal{C}$ *is homogenous.*

Next, we define a transformation of a *PCD* $\mathcal{H}$ to a normal form $NF(\mathcal{H})$ such that $\mathcal{H}$ and $NF(\mathcal{H})$ are equivalent with respect to Lyapunov and asymptotic stability.

Definition 8. *Given* $\mathcal{H} = (\mathcal{C}, F)$, $NF(\mathcal{H}) = (\mathcal{C}', F')$, *where* $\mathcal{C}' \subseteq \mathcal{C}$ *is the set of homogenous predicates in* $\mathcal{C}$, *and* $F'(A) = F(A')$, *where* $A' = A \cup \{C \,|\, C \in \mathcal{C} \backslash \mathcal{C}', \mathbf{0} \in [\![C]\!]\}$.

$NF(\mathcal{H})$ is in normal form by definition. Note that all the region of $NF(\mathcal{H})$ contain $\mathbf{0}$ on their boundary.

Proposition 1. *Let* $\mathcal{H} = (\mathcal{C}, F)$ *be a PCD. Then* $NF(\mathcal{H}) = (\mathcal{C}', F')$ *is a normal PCD such that:*

1. $\mathcal{H}$ *is Lyapunov stable if and only if* $\hat{\mathcal{H}}$ *is Lyapunov stable.*
2. $\mathcal{H}$ *is asymptotically stable if and only if* $\hat{\mathcal{H}}$ *is asymptotically stable.*

6.2 Abstraction: Construction of the Weighted Graph

In this section, we present the abstraction of a *PCD* to a finite weighted graph. The abstraction consists of the faces of the regions of the *PCD* as the vertices, and an edge corresponds to the fact that there exists an execution from one face to the other. In addition, we also track how much "closer" the execution moves towards the origin between the starting and ending points of executions from one face to the other. This "scaling" in the distance to the origin appears as a weight on the corresponding edge. The finite graph is then be analysed for deducing the stability of the original *PCD*.

First, we need the definition of a region execution.

Definition 9. *Given a region* R *of* $\mathcal{H}$, *an* R-execution *of* $\mathcal{H}$ *is an execution* σ *of* $\mathcal{H}$ *such that* $\sigma(t) \in R$ *for every* $t \in (0, Size(Dom(\sigma)))$. *An* R*-execution* σ *of* $\mathcal{H}$ *is said to be* extreme *if it is finite and* $\sigma(0) \in f_1$ *and* $\sigma(Size(Dom(\sigma))) \in f_2$ *for some* $f_1, f_2 \in Faces(R)$. *A* region execution *of* $\mathcal{H}$ *is an* R-execution *of* $\mathcal{H}$ *for some region* R *of* $\mathcal{H}$.

Remark 1. Note that every region execution is either a finite execution or a complete execution.

Abstract Graph $G(\mathcal{H})$ and Its Correctness. Let $\mathcal{H} = (\mathcal{C}, F)$ be an n-dimensional *PCD*. The weighted graph $G(\mathcal{H}) = (V, E, W)$ is defined as follows.

1. The set of vertices V is the set of faces of the regions of $\mathcal{H}$, that is, $f \in Faces(R)$, where $R \in Regions(\mathcal{C})$.
2. The set of edges E consists of pairs (f_1, f_2) such that $f_1, f_2 \in Faces(R)$ for some $R \in Regions(\mathcal{C})$ and there exists a finite R-execution $\sigma \in Exec(\mathcal{H})$ with $\sigma(0) \in f_1$, $\sigma(Size(Dom(\sigma))) \in f_2$.
3. The weight associated with the edge $e = (f_1, f_2) \in E$ is given by $\sup\{|\sigma(Size(Dom(\sigma)))|/|\sigma(0)| : \sigma$ is a finite region execution of $\mathcal{H}, \sigma(0) \in f_1, \sigma(0) \neq \mathbf{0}, \sigma(Size(\ Dom(\sigma))) \in f_2\}$.

Note that the weight on all the edges is greater than or equal to 0. The next proposition states that the weight on an edge is an upper bound on the scaling of all the executions corresponding to it.

Proposition 2. *A finite R-execution σ of $\mathcal{H}$ with $\sigma(0) \in f_1$ and $\sigma(Size(Dom(\sigma))) \in f_2$ for some $f_1, f_2 \in Faces(R)$ is such that $Scaling(\sigma) \leq W((f_1, f_2))$.*

The next theorem states a sufficient condition on the graph $G(\mathcal{H})$ for the *PCD* $\mathcal{H}$ to be Lyapunov stable. We need the definition of an exploding region.

Definition 10. *An execution σ of $\mathcal{H}$ is said to be* diverging *if for every $\epsilon > 0$, there exists a $t \in Dom(\sigma)$ such that $\sigma(t) \notin B_\epsilon(\mathbf{0})$.*

Definition 11. *A region R of a PCD $\mathcal{H}$ is said to be* exploding *if there exists an R-execution σ such that σ diverges.*

Proposition 3. *A region R of a PCD $\mathcal{H}$ is exploding if and only if $\varphi(R) \in Cone(R, \mathbf{0})$.*

Theorem 1. *A PCD $\mathcal{H}$ is Lyapunov stable if the following hold:*

1. *Every simple cycle π of $G(\mathcal{H})$ satisfies $W(\pi) \leq 1$.*
2. *No region of $\mathcal{H}$ is exploding.*

The above theorem says that the abstraction using the finite weighted graph is sound. In fact, for the two dimensional case the theorem provides a necessary and sufficient condition (see [25]). However, the same is not true for higher dimensions.

Theorem 2. *A PCD $\mathcal{H}$ is asymptotically stable if the following hold:*

1. *Every simple cycle π of $G(\mathcal{H})$ satisfies $W(\pi) < 1$.*
2. *No region of $\mathcal{H}$ is exploding.*

Computing the Graph. Next, we explain the algorithm for computing the edges and the weights associated with them.

Given a region $R \in \mathit{Regions}(\mathcal{H})$ and faces $f_1, f_2 \in \mathit{Faces}(R)$, let $W_{Reg}(f_1, f_2, R)$ denote $\sup\{|\sigma(\mathit{Size}(\mathit{Dom}(\sigma)))|/|\sigma(0)| \mid \sigma \text{ is a finite } R\text{-execution of } \mathcal{H}, \sigma(0) \in f_1, \sigma(0) \neq \mathbf{0}, \sigma(\mathit{Size}(\mathit{Dom}(\sigma))) \in f_2\}$. Note that $W((f_1, f_2)) = max\{ W_{Reg}(f_1, f_2, R) \mid R \in \mathit{Regions}(\mathcal{H}), f_1, f_2 \in \mathit{Faces}(R)\}$. Next we show how to compute $W_{Reg}(f_1, f_2, R)$ by using linear programming.

Let us fix a region R and faces $f_1, f_2 \in \mathit{Faces}(R)$. First, observe that $W_{Reg}(f_1, f_2, R)$ is equal to

$$\sup\{|\mathbf{v}_2|/|\mathbf{v}_1| \mid \exists t \geq 0, \mathbf{v}_1 \in f_1, \mathbf{v}_1 \neq \mathbf{0}, \mathbf{v}_2 \in f_2, \mathbf{v}_2 = \mathbf{v}_1 + \varphi(R)t\} \tag{1}$$

Let $\psi(f_1, f_2, R) = \exists t \geq 0, \mathbf{v}_1 \in f_1, \mathbf{v}_1 \neq \mathbf{0}, \mathbf{v}_2 \in f_2, \mathbf{v}_2 = \mathbf{v}_1 + \varphi(R)t$. If $\psi(f_1, f_2, R)$ is feasible, then Equation 1 is equivalent to the following:

$$\sup |\mathbf{v}_2|/|\mathbf{v}_1| s.t. \tag{2}$$
$$t \geq 0, \mathbf{v}_1 \in \mathit{Closure}(f_1), \mathbf{v}_1 \neq \mathbf{0}, \mathbf{v}_2 \in \mathit{Closure}(f_2), \mathbf{v}_2 = \mathbf{v}_1 + \varphi(R)t$$

Further, we observe that it suffices to consider $\mathbf{v}_1$ such that $|\mathbf{v}_1| = 1$, due to the fact that the constraints corresponding to the faces and the regions are homogenous linear constraints. Hence Equation 2 is equivalent to:

$$\sup |\mathbf{v}_2| s.t. \tag{3}$$
$$t \geq 0, \mathbf{v}_1 \in \mathit{Closure}(f_1), \mathbf{v}_2 \in \mathit{Closure}(f_2), \mathbf{v}_2 = \mathbf{v}_1 + \varphi(R)t, |\mathbf{v}_1| = 1$$

Note that Equation 3 is not a linear program in general. However, observe that the results in the paper do not depend on the norm. Hence, we can choose the infinity norm for our analysis, which is defined as, $|\mathbf{x}| = max_i \langle \mathbf{x} \rangle_i$. Then we can computing Equation 3 by solving $O(n^2)$ linear programs, where n is the dimension of $\mathcal{H}$, as follows. Define a linear program $P(i, j, \alpha, \beta)$, where $1 \leq i, j \leq n$ and $\alpha, \beta \in \{-1, +1\}$ as follows:

$$max\, \alpha \langle \mathbf{y} \rangle_i s.t. \tag{4}$$
$$t \geq 0, \mathbf{x} \in \mathit{Closure}(f_1), \mathbf{y} \in \mathit{Closure}(f_2), \mathbf{y} = \mathbf{x} + \varphi(R)t$$
$$\alpha \langle \mathbf{y} \rangle_i \geq 0, \langle \mathbf{x} \rangle_j = \beta,$$

and for every $1 \leq k \leq n, k \neq j$

$$-1 \leq \mathbf{x}_k \leq 1$$

Note that for a fixed i, j, α and β, $P(i, j, \alpha, \beta)$ is a linear programming problem. Further, Equation 3 is equivalent to $max_{i,j,\alpha,\beta} P(i, j, \alpha, \beta)$.

The following lemma summarizes the computation of the weights on the edges, and show that solving Equation 3 is equivalent to solving $4n^2$ linear programming problems.

Lemma 1. *If $\psi(f_1, f_2, R)$ holds, then*

$$W_{Reg}(f_1, f_2, R) = max_{1 \leq i,j \leq n, \alpha, \beta \in \{-1,+1\}} P(i, j, \alpha, \beta).$$

6.3 Model-Checking

In this section, we discuss the algorithm for verifying the conditions in Theorem 1 and Theorem 2.

First, note that in order to check if a region R is exploding, using Proposition 3, it suffices to check if $\varphi(R) \in \mathit{Cone}(R, \mathbf{0})$. This condition is equivalent to checking if $t > 0 \wedge \varphi(R)t \in R$ is satisfiable, which can be verified using an SMT solver for linear real arithmetic.

Next, we describe a dynamic programming solution for analyzing the graph for cycles with weight greater than or equal to 1. It is similar to the the dynamic programming solution to the shortest path algorithm. However, the operation of addition is replaced by multiplication here. We compute iteratively $g^k(f_1, f_2)$ for each $1 \leq k \leq s$, where s is number of vertices in the graph and $g^k(f_1, f_2)$ is the maximum weighted path among all paths from f_1 to f_2 in $G(\mathcal{H}) = (V, E, W)$ of length at most k. The iterative computation is as follows:

1. $g^1(f_1, f_2) = W(f_1, f_2)$ if $(f_1, f_2) \in E$ and -1 otherwise.
2. $g^k(f_1, f_2) = max\{g^{k-1}(f_1, f_2), max_{(f_2', f_2) \in E}(g^{k-1}(f_1, f_2') \times W(f_2', f_2))\}$, for $1 < k \leq s$.

The next lemma states that the maximum weight of the paths in the graph can be used to analyse for existence of cycles of weight greater than or equal to 1.

Lemma 2. *Let g be the function computed above, and s be the number of vertices in $G(\mathcal{H})$. Then:*

1. *$g^s(f, f) \leq 1$ for every $f \in V$ if and only if $G(\mathcal{H})$ does not contain simple cycles with weight > 1.*
2. *$g^s(f, f) < 1$ for every $f \in V$ if and only if $G(\mathcal{H})$ does not contain simple cycles with weight ≥ 1.*

The above lemma relies on the fact that if there exists a cycle of weight > 1, then at least one of the cycles in its simple cycle decomposition has weight > 1. The above algorithm takes time polynomial in the number of vertices or equivalently faces. One can in fact trace the cycles which violates the condition of Theorem 1 or Theorem 2. Such a cycle is a *counter-example* in the abstract weighted graph which shows a potential violation of stability.

7 Experiments

Our abstraction and refinement algorithms have been implement in Python 2.7 and the experiments are run on Mac OS X 10.5.8 with a 2 GHz Inter Core 2 Duo processor and a 4GB 1067 MHz DDR3 memory. We use the SMT solver Z3 version 4.3.2 to solve satisfiability of linear real arithmetic formulas which are required for determining the existence of edges and checking if a region is exploding. We use the linear programming package GLPK version 4.8 for solving the linear optimization problem required in constructing the weights on the edges. There are no standard benchmarks for evaluating the stability of hybrid systems, especially, for the class that we consider. Hence, most of the experiments are performed on hand constructed examples. Many of them are generalizations of the three dimensional example considered in Section 2 by starting with an n-dimensional grid and transforming it systematically into an example in a normal

form of $n+1$-dimensions. We believe that by extending the approach to more complex hybrid systems, we can experiment on examples from real applications.

Our results are tabulated in Table 1. In the table, the column with heading *Var* corresponds to the number of variables in the *PCD*, *Pred* corresponds to the number of predicates, *Reg* to the number of regions, *Faces* to the number of faces, *FT*, *GT*, *VT* and *Total* are the time in seconds for constructing the faces, the weighted graph, the verification (graph analysis for cycles) and the total time, *Lyap* states whether the system is Lyapunov stable or not, *Asymp* states whether the system is asymptotically stable or not, and *GL* and *GA* report the result of the abstraction based analysis stating whether the abstract system is Lyapunov stable and asymptotically stable respectively.

Table 1. Experiments

Var	*Pred*	*Reg*	*Faces*	*FT(sec)*	*GT* (sec)	*VT* (sec)	*Total* (sec)	*Lyap*	*GL*	*Asymp*	*GA*
2	1	8	9	2.3	1.8	< 1	5	*Y*	*Y*	*Y*	*Y*
2	2	12	13	5.7	2.6	< 1	9	*Y*	*Y*	*Y*	*Y*
2	3	16	17	10.2	3.8	< 1	15	*Y*	*Y*	*Y*	*Y*
2	4	20	21	15.2	4.5	< 1	21	*Y*	*Y*	*Y*	*Y*
2	5	24	25	21.7	5.6	< 1	28	*Y*	*Y*	*Y*	*Y*
3	1	32	75	36.7	62.4	< 1	100	*Y*	*N*	*N*	*N*
3	2	72	179	104.1	156.9	38	299	*Y*	*N*	*N*	*N*
3	3	128	331	130.7	306.7	575	1013	*Y*	*N*	*N*	*N*

We consider a two dimensional and a three dimensional example, and experiment with an increasing set of predicates. The face construction and the graph construction time increase almost linearly in the number of faces. The graph analysis time is at most cubic in the size of the graph. However, the number of faces can increase quickly with the increase in the number of predicates. Hence, it is crucial to choose the predicates carefully. Also, observe that there are cases where the system is stable, but the graph construction is too coarse and hence concludes that the system is unstable. Adding more predicates might prove that the system is stable. In this paper, we do not focus on the problem of choosing the right predicates. The future work will focus on this problem by incorporating an abstraction refinement loop to the algorithm.

8 Conclusions

In this paper, we explored model-checking as an alternate approach to stability verification. We proposed a modification of predicate abstraction based on faces in the system to abstract into a finite weighted graph, where the weights track the increase or decrease in the distance of the state along the executions, which can be efficiently constructed using linear programming. The graph was then verified for the absence of simple cycles whose weight is less than or equal to

1 for Lyapunov stability and less than 1 for asymptotic stability, which can be efficiently solved using dynamic programming. Our experimental results show that the method has the potential to scale.

Our model-checking algorithm returns an abstract counter-example which is a simple cycle with weight greater than 1 for Lyapunov stability and weight greater than or equal to 1 for asymptotic stability. One interesting future direction is to develop a counter-example guided abstraction refinement framework using the abstract counter-example returned in the model-checking phase. Another research direction is to extend the results to more complex dynamics. The extension of our results to the case of hybrid systems with polyhedral constraints (not necessarily non-overlapping invariants) and constant or rectangular flows should be straightforward. More complex classes might require additional work given that the exact post computation is not feasible in general.

Acknowledgements. We would like to thank Mahesh Viswanathan for discussions on the topic.

References

1. Alur, R., Dang, T., Ivan**v**ci´c, F.: Counter-Example Guided Predicate Abstraction of Hybrid Systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
2. Asarin, E., Dang, T., Girard, A.: Hybridization methods for the analysis of nonlinear systems. Acta Informatica 43(7), 451–476 (2007)
3. Asarin, E., Maler, O., Pnueli, A.: Reachability analysis of dynamical systems having piecewise-constant derivatives. Theoretical Computer Science 138(1), 35–65 (1995)
4. Branicky, M.S.: Stability of hybrid systems. In: Unbehauen, H. (ed.) Encylopedia of Life Support Systems. Control Sytems, Robotics and Automation, volume Theme 6.43, chapter Article 6.43.28.3. UNESCO Publishing (2004)
5. Chen, X., Abraham, E., Sankaranarayanan, S.: Taylor model flowpipe construction for non-linear hybrid systems. In: Proceedings of the IEEE Real-Time Systems Symposium (2012)
6. Clarke, E.M., Fehnker, A., Han, Z., Krogh, B., Ouaknine, J., Stursberg, O., Theobald, M.: Abstraction and Counterexample-Guided Refinement in Model Checking of Hybrid Systems. International Journal on Foundations of Computer Science 14(4), 583–604 (2003)
7. Davrazos, G., Koussoulas, N.T.: A review of stability results for switched and hybrid systems. In: Proceedings of the Mediterranean Conference on Control (2001)
8. Dierks, H., Kupferschmid, S., Larsen, K.G.: Automatic Abstraction Refinement for Timed Automata. In: Proceedings of Formal Modeling and Analysis of Timed Systems, pp. 114–129 (2007)
9. Duggirala, P.S., Mitra, S.: Lyapunov abstractions for inevitability of hybrid systems. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control, pp. 115–124 (2012)
10. Frehse, G.: PHAVer: Algorithmic verification of hybrid systems past hytech. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 258–273. Springer, Heidelberg (2005)

11. Girard, A.: Reachability of uncertain linear systems using zonotopes. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 291–305. Springer, Heidelberg (2005)
12. Goebel, R., Sanfelice, R., Teel, A.: Hybrid dynamical systems. IEEE Control Systems, Control Systems Magazine 29, 28–93 (2009)
13. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? In: Proceedings of the ACM Symposium on Theory of Computation, pp. 373–382 (1995)
14. Henzinger, T.A.: The Theory of Hybrid Automata. In: Proceedings of the IEEE Symposium on Logic in Computer Science, pp. 278–292 (1996)
15. Khalil, H.K.: Nonlinear Systems. Prentice-Hall, Upper Saddle River (1996)
16. Liberzon, D.: Switching in Systems and Control. Birkhuser, Boston (2003)
17. Oehlerking, J., Burchardt, H., Theel, O.: Fully automated stability verification for piecewise affine systems. In: Bemporad, A., Bicchi, A., Buttazzo, G. (eds.) HSCC 2007. LNCS, vol. 4416, pp. 741–745. Springer, Heidelberg (2007)
18. Parrilo, P.A.: Structure Semidefinite Programs and Semialgebraic Geometry Methods in Robustness and Optimization. PhD thesis, California Institute of Technology, Pasadena, CA (May 2000)
19. Platzer, A., Quesel, J.-D.: KeYmaera: A hybrid theorem prover for hybrid systems (system description). In: Armando, A., Baumgartner, P., Dowek, G. (eds.) IJCAR 2008. LNCS (LNAI), vol. 5195, pp. 171–178. Springer, Heidelberg (2008)
20. Podelski, A., Wagner, S.: Model checking of hybrid systems: From reachability towards stability. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 507–521. Springer, Heidelberg (2006)
21. Prabhakar, P., Vladimerou, V., Viswanathan, M., Dullerud, G.E.: Verifying tolerant systems using polynomial approximations. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 181–190 (2009)
22. Prabhakar, P., Duggirala, P.S., Mitra, S., Viswanathan, M.: Hybrid automata-based CEGAR for rectangular hybrid systems. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 48–67. Springer, Heidelberg (2013)
23. Prabhakar, P., Dullerud, G.E., Viswanathan, M.: Pre-orders for reasoning about stability. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control, pp. 197–206 (2012)
24. Prabhakar, P., Viswanathan, M.: A dynamic algorithm for approximate flow computations. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control, pp. 133–143 (2010)
25. Prabhakar, P., Viswanathan, M.: On the decidability of stability of hybrid systems. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control (2013)
26. Prajna, S., Jadbabaie, A.: Safety verification of hybrid systems using barrier certificates. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 477–492. Springer, Heidelberg (2004)
27. Puri, A., Borkar, V.S., Varaiya, P.: ϵ-approximation of differential inclusions. In: Proceedings of the International Conference on Hybrid Systems: Computation and Control, pp. 362–376 (1995)

System Level Formal Verification via Model Checking Driven Simulation

Toni Mancini, Federico Mari, Annalisa Massini, Igor Melatti, Fabio Merli, and Enrico Tronci

Computer Science Department - Sapienza University of Rome
Via Salaria 113, I-00198 Roma, Italy
{tmancini,mari,massini,melatti,merli,tronci}@di.uniroma1.it

Abstract. We show how by combining *Explicit Model Checking* techniques and simulation it is possible to effectively carry out (bounded) *System Level Formal Verification* of large *Hybrid Systems* such as those defined using *model-based* tools like *Simulink*.

We use an *explicit model checker* (namely, *CMurphi*) to generate all possible (*finite horizon*) simulation scenarios and then optimise the simulation of such scenarios by exploiting the ability of simulators to *save* and *restore* visited states. We show feasibility of our approach by presenting experimental results on the verification of the fuel control system example in the Simulink distribution. To the best of our knowledge this is the first time that (exhaustive) verification has been carried out for hybrid systems of such a size.

1 Introduction

System Level Verification of Embedded Systems has the goal of verifying that the *whole* (i.e., software + hardware) system meets the given specifications. Hardware In the Loop Simulation (HILS) is currently the main workhorse for system level verification and is supported by *Model Based Design* tools like Simulink (`http://www.mathworks.com`) and VisSim (`http://www.vissim.com`). In HILS the *actual software* reads [sends] values from [to] mathematical models (*simulation*) of the physical systems (e.g. engines, analog circuits, etc.) it will be interacting with.

The main concerns in a HILS campaign are: the effort needed to define the simulation scenarios (may require months of work from expert designers), the time needed to carry out the campaign itself (may require weeks or even months of simulation activity), the degree of *assurance* achieved at the end of the HILS campaign. In this paper we show how using Explicit Model Checking (EMC) techniques it is possible to advance the state of the art on all points above.

1.1 Main Contributions

Our System Under Verification (SUV) is a *Hybrid System* (e.g., see [1] and citations thereof) whose inputs belong to a finite set of uncontrollable events (*disturbances*) modelling failures in sensors or actuators, variations in the system parameters, etc. We focus

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 296–312, 2013.

on *deterministic systems* (the typical case for control systems), and model nondeterministic behaviours (such as faults) with disturbances. Accordingly, in our framework, a *simulation scenario* is just a finite sequence of disturbances.

A system is expected to *withstand* all disturbance sequences that may arise in its operational scenario. Correctness of a system is thus defined with respect to such *admissible* disturbance sequences. The set of admissible disturbance sequences typically satisfies constraints like the following: 1) the number of failures occurring within a certain period of time is below a given threshold; 2) the time interval between two consecutive failures is greater than a given threshold; 3) a failure is repaired within a certain time, etc. Thus, in our setting, the set of admissible disturbance sequences (*disturbance model*) can be defined using a Finite State Automaton, which in turn can be defined using the modelling language of any finite state model checker. To this end we use CMurphi [10,9] since its rule based modelling language turns out to be quite handy to define admissible disturbance sequences.

In such a framework we address *Bounded* System Level Formal Verification (SLFV) of *safety* properties. That is, given a time horizon T and a time step τ (time quantum between disturbances) we return *PASS* if there is no *admissible* disturbance sequence of length T and time step τ that violates the given safety property. We return *FAIL*, along with a counterexample, otherwise. Therefore, SLFV is an *exhaustive* (with respect to admissible simulation scenarios) HILS. In other words, we are aiming at (*black box*) *bounded model checking* where the SUV behaviour is defined by a simulator (Simulink in our examples).

To enable an effective parallel approach to SLFV, we split the verification process into two main phases. First, an *off-line* phase, where Explicit Model Checking techniques are used to compute, from the disturbance model, say k, highly optimised simulation campaigns for a set of k simulators. Second, an *on-line* parallel phase where each simulator runs its simulation campaign independently and stops as soon as an error is found. The rationale is that the simulation phase is the heavier one from a computational point of view, thus our approach aims at parallelising such a phase.

Note that if an error is found, only the *on-line* phase above has to be repeated since the set of admissible simulation scenarios computed in the *off-line* phase does not change. The *on-line* phase is supported by simulation tools (Simulink in our examples). Here we provide methods and tools to effectively carry out the *off-line* phase computing *optimised* simulation campaigns for the available simulator.

While most *Model Based Testing* approaches focus on modelling the SUV, in this work we model the set of disturbances the SUV is supposed to withstand. Accordingly, the performance of our *off-line* phase does not depend on the SUV model and only depends on the disturbance model. On the other hand, simulation times in the *on-line* phase depend on the size of SUV and disturbance models.

We implemented our approach and present experimental results on its usage on the fuel control system example in the Simulink distribution. In our experiments we set our *time horizon* to 100 seconds and our *time step* to 1 second. Our main contributions can be summarised as follows.

Automatic Exhaustive Simulation Scenario Generation. We show how a suitable search on the transition graph of (the automaton defining) the disturbance model can be

used to generate *all and only* the admissible disturbance sequences (simulation scenarios) split into k disjoint *slices*. Such an initial partitioning of the simulation scenarios allows us to distribute our later steps among k parallel processes. We implemented such an *exhaustive simulation scenario generator* within the CMurphi model checker. Our case study disturbance model yields about 4 million disturbance traces (simulation scenarios) stored into a 3.5GB file. To generate such traces our algorithm takes about 30 minutes and within 15 seconds splits them into $k = 64$ *slices* of equal size.

Simulation Campaign Optimisation. We present a disk based *optimisation* algorithm that transforms a sequence of simulation scenarios into a very efficient *simulation campaign*, that is a sequence of simulation instructions (namely: *save* a simulation state, *restore* a saved simulation state, *inject* a disturbance, *advance* the simulation of a given time length). Our algorithm will be run in parallel on k processors, each taking as input a different slice of the simulation scenarios. For example, when using $k = 8$ [$k = 64$] parallel processors our algorithm can compute k disjoint optimised simulation campaigns for our case study in about 44 minutes [one minute]. Simulation of the optimised campaign for $k = 64$ takes about 3 days, whereas simulation of the *unoptimised* one, that is a simulation campaign that does not exploit the save/restore features of simulators, thereby always restarting scenario simulations from the initial state, takes about 12 days (i.e., the speed-up is about $3.8\times$). Similarly to Explicit Model Checking (e.g., CMurphi [9], SPIN [14]), our simulation campaign optimiser counteracts *scenario explosion* by avoiding as much as possible revisiting already visited simulator states. Since the size of a simulation state can easily take many MB, it is not possible to store too many states, even resorting to the disk. Thus, a clever strategy is needed to decide when to save a visited state or just to recompute it. This is what our simulation campaign optimiser does, thereby transforming a simulator into a sort of *explicit model checker*.

Summing Up. We show how using explicit model checking techniques it is possible to generate optimised simulation campaigns for a set of simulators. This enables parallel HILS based SLFV. We show the effectiveness of the proposed approach on an industrial case study in the Simulink distribution. To the best of our knowledge this is the first time that SLFV is carried out for a *real world* hybrid system of such a size.

1.2 Related Work

The paper closest to ours is [6] where CMurphi capability to call external C functions in a *black box* fashion has been used to drive the ESA satellite simulator SIMSAT in order to verify satellite operational procedures. Along the same line of thinking, in [16] the analogous SPIN capability has been used to verify actual C code. Such approaches differ from ours since optimisation of the simulation campaign is not considered. Safety checking has been widely investigated in a finite state setting (e.g., see [22] and citations thereof). In our setting, *black box* verification of continuous time hybrid systems, we check specifications using *monitors*, similarly to [18].

Statistical model checking, being basically *black box*, is also closely related to our approach. In such a setting, [31] is closely related to our paper since it addresses system level verification of Simulink models and presents experimental results on the very same Simulink case study we are using. Monte Carlo model checking methods

(see, e.g., [23,27,12]) are also related to our approach. The main differences between the above statistical approaches and ours are the following: (i) statistical methods *sample* the space of admissible simulation scenarios, whereas we address *exhaustive* HILS; (ii) statistical methods do not address optimisation of the simulation campaign which is our main concern here, since this is what makes exhaustive HILS viable. It is worth noticing that in trading off *precision* of the answer to the SLFV problem and *size* of the set of admissible scenarios, statistical model checking and our proposed approach are somehow complementary. In fact, the former returns a *statistical* answer but can consider (potentially) infinite sets of admissible scenarios whereas the latter, being exhaustive, returns a *precise* answer, but can only address finite sets of admissible scenarios.

Formal verification of Simulink models has been widely investigated, examples are in [26,19,29]. Such methods however focus on discrete time models (e.g., Stateflow or Simulink restricted to discrete time operators) with small domain variables. Therefore they are well suited to analyse critical subsystems, but cannot handle complex system level verification tasks (e.g., as our case study). This is indeed the motivation for the development of statistical model checking methods as the one in [31] and for our exhaustive HILS based approach.

Of course *Model Based Testing* (e.g., see [5]) has widely considered automatic generation of test cases from models. In our HILS setting, automatic generation of simulation scenarios (for Simulink) has been investigated in [11,17,4,28]. The main differences with our approach are the following. First, such approaches cannot be used in our *black box* setting since they generate simulation scenarios from the Stateflow/Simulink model of the SUV (whereas we generate scenarios from the disturbance model). Second, the above approaches are not exhaustive, whereas ours is.

Synergies between simulation and formal methods have been widely investigated in digital hardware verification. Examples are in [30,13,21,7] and citations thereof. The main differences between the above approaches and ours are: (i) they focus on finite state systems whereas we focus on infinite state systems (namely, hybrid systems); (ii) they are *white box* (requiring availability of the system model) whereas we are *black box*. We note that the idea of speeding up the simulation process by saving and restoring suitably selected visited states is also present in [7].

Parallel algorithms for explicit state exploration have been widely investigated. Examples are in [25,2,20,3,15]. The main difference with our approach is that all the above ones focus on parallelising the state space exploration engine by devising techniques to minimise locking of the visited state hash table whereas we leave unchanged the state space exploration engine (the simulator in our context) and use an embarrassing parallel (*map and reduce like* [8]) strategy that splits (*map* step) the set of simulation scenarios into equal size subsets to be simulated on different processors and stops verification as soon as one of such processors finds an error (*reduce* step).

1.3 Outline of the Paper

Section 2 defines how we model disturbances, SUV, and our SLFV problem. Section 3 formalises the notion of simulator. Section 4 outlines how disturbance traces are generated from a CMurphi model. Section 5 outlines our simulation campaign optimisation

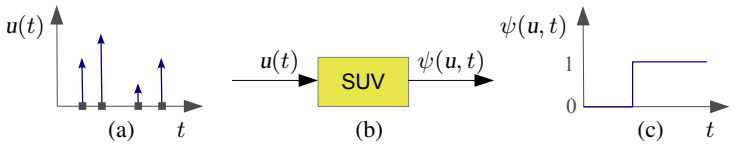


Fig. 1. (a) a discrete event sequence $u \in \mathfrak{U}_d$; (b) our SUV; (c) the SUV output $\psi(u, t)$

algorithm and proves its correctness. Section 6 outlines how we execute simulation campaigns on Simulink and presents experimental results.

2 Bounded System Level Formal Verification

In this section we define the system level verification problem we address (Definition 7). To this end we model disturbances (Definition 1), our SUV (Definitions 2, 3, 4) as well as the class of disturbances (Definitions 5, 6) our SUV is supposed to withstand in its operational scenario.

Throughout the paper, we use $\mathbb{R}^{\geq 0}$, the set of non-negative reals, to represent time, $\mathbb{R}^{+}$, the set of strictly positive reals, to represents non-zero time durations, and Bool $= \{0, 1\}$ to represent Booleans. $\mathbb{N}^{+}$ is the set of positive natural numbers.

A *discrete event sequence* (Definition 1(a) and Fig. 1a), is a function associating to each (continuous) time instant a disturbance event (such as a fault, a variation in system a parameter, etc). As no system can withstand an infinite number of disturbances within a finite time, we require that, in any time interval of finite length, only a finite number of disturbances can take place. We represent with the integer 0 the event carrying no disturbance and with positive integers actual disturbances. Thus we have that, in any finite time interval, a discrete event sequence differs from 0 only in a finite number of time points. An *event list* (Definition 1.b) provides an explicit representation for a discrete event sequence by listing event/time distance pairs for disturbance events.

Definition 1 (Discrete event sequence and event list). *Let* $d \in \mathbb{N}^{+}$.

(a) A discrete event sequence *over integer interval* $[0, d]$ *is a function* $\mathsf{u} : \mathbb{R}^{\geq 0} \rightarrow [0, d]$ *such that, for all* $t \in \mathbb{R}^{\geq 0}$, *the set* $\left\{\tilde{t} \mid 0 \leq \tilde{t} \leq t \text{ and } \mathsf{u}(\tilde{t}) \neq 0\right\}$ *has finite cardinality. Following control engineering notation for input functions to dynamical systems (e.g., see [24]), we denote with* $\mathfrak{U}_d$ *the set of discrete event sequences over* $[0, d]$.

(b) An event list *on* $[0, d]$ *is a (finite or infinite) sequence of pairs:* $(u_0, \tau_0), (u_1, \tau_1), \ldots$ *such that for all* $i \geq 0$, $u_i \in [0, d]$ *and* $\tau_i \in \mathbb{R}^{+}$. *Each event list denotes a unique discrete event sequence* $\mathsf{u}(t)$ *defined as follows:* $\mathsf{u}(0) = u_0$ *and, for each* $t > 0$, *if there exist an integer* $h \geq 0$ *such that* $t = \sum_{i=0}^{h} \tau_i$ *and* (u_{h+1}, τ_{h+1}) *is in the event list, then* $\mathsf{u}(t) = u_{h+1}$, *else* $\mathsf{u}(t) = 0$.

In our setting the system to be verified can be modelled as a Discrete Event System (Definition 2 and Fig. 1b), that is, a continuous time *Input-State-Output* deterministic dynamical system [24] whose input functions are discrete event sequences, whose state can undertake continuous as well as discrete changes, and whose output ranges on any combination of discrete and continuous values.

Definition 2 (Discrete Event System). *A Discrete Event System (DES) is a tuple* $\mathcal{H} = (S, s_0, d, O,$ *flow*, *jump*, *output) where:*

– S *is a set of* states *(finite, countable, continuous, or any combination thereof).*

– $s_0 \in S$ *is the* initial state.

– $d \in \mathbb{N}^+$ *defines the* input space *as* $\mathfrak{U}_d$ *(the set of discrete event sequences over* $[0, d]$*).*

– O *is the set of* output *values (finite, countable, continuous, or any combination thereof).*

– *flow* : $S \times \mathbb{R}^{\geq 0} \to S$*. For all* $s \in S$, $t \in \mathbb{R}^{\geq 0}$, *flow*$(s, t)$ *defines the state reached by* $\mathcal{H}$ *from state* s *after time* t *when no event occurs. Accordingly, we stipulate that for all* $s \in S$, *flow*$(s, 0) = s$*.*

– *jump* : $S \times [0, d] \to S$*. For all* $s \in S$, $e \in [0, d]$ *jump*(s, e) *defines the state reached by* $\mathcal{H}$ *from state* s *upon occurrence of event* e *(no time flows). Accordingly, we stipulate that for all* $s \in S$, *jump*$(s, 0) = s$*.*

– *output* : $S \to O$*. The value output*(s) *defines the output of* $\mathcal{H}$ *in state* s*.*

The state, respectively output, reached after time t by a DES with a given input can be computed with the DES *state*, respectively *output*, function (Definition 3, Fig. 1c).

Definition 3 (DES state and output function). *The* state function *of DES* $\mathcal{H}$ *is a function* $\phi : \mathfrak{U}_d \times \mathbb{R}^{\geq 0} \to S$*, where* $\phi(\mathsf{u}, t)$ *is the state reached at time* t *by* $\mathcal{H}$ *with input the discrete event sequence* u*. Function* ϕ *is defined inductively as follows:*

– $\phi(\mathsf{u}, 0) =$ *jump*$(s_0, \mathsf{u}(0))$*, where* s_0 *is the initial state of* $\mathcal{H}$*;*

– *For each* $t > 0$, $\phi(\mathsf{u}, t) =$ *jump*$($*flow*$(\phi(\mathsf{u}, t^*), t - t^*), \mathsf{u}(t))$*, where:* $t^* < t$ *is the greatest value such that* $\mathsf{u}(t^*) \neq 0$ *and we let* $t^* = 0$ *if such a value does not exist (i.e., when* u *is identically 0 before* t*).*

The output function *of* $\mathcal{H}$ *is the function* $\psi : \mathfrak{U}_d \times \mathbb{R}^{\geq 0} \to O$ *defined as* $\psi(\mathsf{u}, t) =$ *output*$(\phi(\mathsf{u}, t))$*. In other words,* ψ *computes the output (as a function of time) of* $\mathcal{H}$ *when the input to* $\mathcal{H}$ *is the discrete event sequence* u*. In general,* $\psi(\mathsf{u}, t)$ *is not a discrete event sequence (e.g., it may take a non-zero value an infinite number of times).*

We model the property to be verified with a continuous-time *monitor* that observes the state of the system to be verified and checks whether the property under verification is satisfied. Thus we can handle any property for which a monitor exists. In particular bounded safety and bounded liveness properties can easily be modelled using monitors. Checking properties with Simulink monitors can be done as outlined in [18].

Since we observe our monitor output only at discrete time points, we may miss a property failure report. To avoid this, we ask our monitor output to be 0 as long as the property under verification is satisfied and to become and stay 1 (*sustain*) as soon as the property fails. Since the monitor output is all we need to carry out our verification task, we model our System Under Verification as a DES with an embedded monitor whose set of output values is Bool. We call Monitored DES such a DES (Definition 4, summarised in Fig. 1).

Definition 4 (Monitored DES). *A Monitored DES (MDES) is a tuple* $\mathcal{H} = (S, s_0,$ $d,$ *flow*, *jump*, *output) such that* $(S, s_0, d,$ *Bool*, *flow*, *jump*, *output) is a DES whose output function* $\psi(\mathsf{u}, t)$ *is non-decreasing with respect to* t*. That is, for any input sequence*

$\mathfrak{u} \in \mathfrak{U}_d$*, for all* $t, t' \in \mathbb{R}^{\geq 0}$*, if* $t \leq t'$ *then* $\psi(\mathfrak{u}, t) \leq \psi(\mathfrak{u}, t')$*. In other words, an MDES is a DES with non-decreasing boolean outputs.*

Admissible disturbance sequences (or *traces*) formally model the set of operational scenarios our SUV is supposed to *withstand*. It is typically infeasible to explicitly list all such scenarios manually. Therefore, in our *Model Based* approach we define (Definition 5) them with a suitable finite state automaton with *guarded* transitions and initial as well as final (accepting) states. An (admissible) disturbance trace is then a sequence of transitions from an initial to a final state (Definition 6). In our setting, guards (*adm* in Definition 5) are used to define the set of disturbances that may occur in a given state, whereas final states are used to model constraints on whole disturbance traces. For example, if our set of disturbance traces consists of traces where a certain disturbance event (say, d_1) only occurs at most three times but never immediately after disturbance d_2, then we can use guard *adm* to disable occurrence of d_1 immediately after d_2 and take as final states those where d_1 has occurred at most three times.

Definition 5 (Disturbance generator). *A Disturbance Generator (DG) is a tuple* $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ *where:*

- Z *is a finite set of states.*
- $Z_I \subseteq Z$ *and* $Z_F \subseteq Z$ *are the set of, respectively, initial and final states.*
- $d \in \mathbb{N}^+$ *defines the set of disturbance events represented (without loss of generality) with integers in* $[0, d]$*, where value 0 represents the event carrying no disturbance.*
- $\mathsf{dist} : Z \times [0, d] \rightarrow Z$ *is a (*deterministic transition*) function mapping each state/disturbance pair* (z, e) *to a* next *state* $\mathsf{dist}(z, e)$*.*
- $\mathsf{adm} : Z \times [0, d] \rightarrow \mathit{Bool}$ *is a (*guard*) function defining (*the characteristic function of*) the set of disturbances admissible (i.e., that may occur) in a given state.*

A disturbance generator, being a finite state automaton, can be defined using the input language of any finite state model checker. Note that we model simultaneous disturbances as one single event (i.e., one disturbance). Definition 6 defines disturbance traces (simulation scenarios) as paths from initial to final states in a DG. Since we are in a *Bounded Model Checking* setting, we focus on disturbance traces of finite length.

Definition 6 (Disturbance trace and associated event list). *Let* $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ *be a DG.*

(a) A disturbance path *of length* h *for* $\mathcal{D}$ *is a computation path in* $\mathcal{D}$ *with* h *disturbances (transitions). Formally, a* disturbance path *of length* h *for* $\mathcal{D}$ *is a sequence* $z_0, d_0,\ z_1, d_1, \ldots, z_{h-1}, d_{h-1},\ z_h$*, where* $z_0 \in Z_I$*,* $z_h \in Z_F$ *and, for all* $0 \leq i < h$*,* $z_i \in Z$*,* $d_i \in [0, d]$*,* $\mathsf{adm}(z_i, d_i) = 1$*, and* $z_{i+1} = \mathsf{dist}(z_i, d_i)$*.*

(b) A disturbance trace *of length* h *for* $\mathcal{D}$ *is a sequence* $\delta = d_0, \ldots, d_{h-1}$ *of* h *disturbances such that there exists a disturbance path* $z_0, d_0,\ z_1, d_1, \ldots, z_{h-1}, d_{h-1},\ z_h$ *for* $\mathcal{D}$*.*

(c) Given a time step $\tau \in \mathbb{R}^+$*, the* event list associated to a disturbance trace $\delta = d_0, \ldots, d_{h-1}$ *with respect to* τ *evenly maps the events in* δ *on the time axis at time points multiple of* τ*. Formally, the* event list associated *to* δ *with respect to a time step* $\tau \in \mathbb{R}^+$ *is* $\mathfrak{u}_\tau(\delta) = (d_0, \tau), \ldots, (d_{h-1}, \tau)$*.*

(d) We denote with $\Delta_{\mathcal{D}}^h$ *the* set of all disturbance traces of length h for $\mathcal{D}$.

System Level Formal Verification (SLFV) (Definition 7) aims at verifying that our SUV (modelled as an MDES) can *withstand* all disturbance traces (defined with a DG) that may occur in the SUV operational environment.

Definition 7 (System Level Formal Verification problem). *A System Level Formal Verification (SLFV) problem is a tuple* $(\mathcal{H}, \mathcal{D}, \tau, h)$*, where:* $\mathcal{H} = (S, s_0, d, \mathsf{flow}, \mathsf{jump}, \mathsf{output})$ *is an MDES (modelling our SUV),* $\mathcal{D} = (Z, d, \mathsf{dist}, \mathsf{adm}, Z_I, Z_F)$ *is a DG (modelling our SUV operational scenario and whose set of outputs* $[0, d]$ *is equal to the set of input values of* $\mathcal{H}$*),* $\tau \in \mathbb{R}^+$ *is a* time step *(for disturbance occurrences), and* $h \in \mathbb{N}^+$ *is a* horizon *(for our error search).*

Let ψ *be the output function (Definition 3) of* $\mathcal{H}$*. The* answer *to a SLFV problem* $(\mathcal{H}, \mathcal{D}, \tau, h)$*, denoted by* $\mathcal{H}(\Delta^h_{\mathcal{D}})$*, is:*

– *FAIL if there exists* $\delta \in \Delta^h_{\mathcal{D}}$ *(*counterexample*) such that* $\psi(\mathsf{u}_\tau(\delta), \tau h) = 1$ *(i.e., the MDES modelling our SUV signals an error by outputting 1 when given as input the discrete event sequence associated to* δ*);*

– *PASS otherwise (i.e., for all* $\delta \in \Delta^h_{\mathcal{D}}$*,* $\psi(\mathsf{u}_\tau(\delta), \tau h) = 0$*, as* ψ *is non-decreasing by Definition 4).*

In case the answer is FAIL, an error (witnessed by δ*) exists in the SUV (namely, in its software, in its hardware mathematical model or in their interaction).*

Note that, notwithstanding the fact that the number of states of our SUV is infinite and we are in a continuous time setting, to answer a System Level Formal Verification (SLFV) problem we only need to check a finite number of disturbance traces (Definition 7). This is because we are bounding: (i) our time horizon to $T = \tau h$, (ii) the set of time points at which disturbances can take place, by taking τ as the time quantum among disturbance events. Thus, we should make h *large enough* (as in bounded model checking) and τ *small enough* in order to faithfully model our SUV operational scenario. Indeed, as no physical system can withstand arbitrarily (time) close disturbances, any operational scenario can be modelled with the desired precision by suitably choosing τ and h. On such considerations rests the effectiveness of our approach.

3 Simulators and Simulation Campaigns

In HILS based verification the SUV model (for example, a DES defined using MatLab and Stateflow) runs on a simulator (e.g., Simulink) taking as inputs simulation scenarios (disturbance traces in our formal setting). Because of the huge number (about 4 millions in our case study) of simulation scenarios to be considered for exhaustive HILS, the overall number of simulation steps may be prohibitively large if we simulate each scenario from the initial state of the (SUV) simulator. We counteract such a *scenario explosion* by avoiding as much as possible revisiting already visited simulator states, similarly to Explicit Model Checking algorithms (e.g., CMurphi [9] or SPIN [14]). Unfortunately, in our setting each simulator state can be a quite large file (e.g., about 150 KB in our case study). Thus, a clever strategy is needed to decide when to save a visited state or when to just recompute it. Section 5 shows such a strategy. Here, we formalise the notion of DES simulator (Definition 8) to support design and analysis of such strategies.

Definition 8 (DES simulator). *A* DES simulator *is a tuple* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *where:* $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$ *is a DES;* L *is a set of* labels *(denoting simulator states);* W *is the set of simulator* states*;* $m \in \mathbb{N}^+$ *denotes the maximum number of states the simulator can store.*

Each $w \in W$ *is a tuple* (s, u, M) *where:* $s \in S$ *is a state of* $\mathcal{H}$ *or a distinguished sink state* $\perp$*;* $\mathrm{u} \in \mathfrak{U}_d$ *is an event list;* M *(*simulator memory*) is a set of at most* m *triples* $(l, s, \mathrm{u}_s) \in L \times S \times \mathfrak{U}_d$*, such that, for each* $l \in L$*, there exists at most one triple* $(l, s, \mathrm{u}_s) \in M$ *where* l *occurs. The simulator initial state is* $w_0 = (s_0, \emptyset, \emptyset)$.

Each triple in the simulator memory M binds a label $l \in L$ to a state $s \in S$ of $\mathcal{H}$ and to an event list u_s. Definition 9 gives the semantics of simulator commands.

Definition 9 (Simulator commands and transition function). *Let* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *be a DES simulator, with* $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$*.*
The commands for $\mathcal{S}$ *are:* $\mathsf{load}(l)$*,* $\mathsf{store}(l)$*,* $\mathsf{free}(l)$*,* $\mathsf{run}(e, t)$*, where* $l \in L$ *is a label,* $t \in \mathbb{R}^+$ *is a time duration, and* $e \in [0, d]$ *is an event (*l, t, e *are command arguments).*
The transition function *of* $\mathcal{S}$*,* $\mathsf{sim}_{\mathcal{S}}$*, defines how the internal state of* $\mathcal{S}$ *changes upon execution of a command. Namely:* $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{cmd}(\mathsf{args})) = (s', \mathrm{u}', M')$ *when* $\mathcal{S}$ *moves from state* (s, u, M) *to* (s', u', M') *upon processing* cmd *with arguments* args*.*

The function is defined as follows:

- $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{load}(l)) = (s', \mathrm{u}', M)$*, if* $s \neq \perp$ *and* $(l, s', \mathrm{u}') \in M$.
- $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{free}(l)) = (s, \mathrm{u}, M \setminus \{(l, s', \mathrm{u}')\})$*, if* $s \neq \perp$ *and* $(l, s', \mathrm{u}') \in M$.
- $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{store}(l)) = (s, \mathrm{u}, M \cup \{(l, s, \mathrm{u})\})$*, if* $s \neq \perp$*,* $|M| < m$ *and* $\neg\exists s', \mathrm{u}'[(l, s', \mathrm{u}') \in M]$.
- $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{run}(e, t)) = (s', \mathrm{u}', M)$*, where* $s \neq \perp$*,* $s' = \mathit{flow}(\mathit{jump}(s, e), t)$*, and* u' *is* (e, t) *concatenated to* u*.*
- $\mathsf{sim}_{\mathcal{S}}(s, \mathrm{u}, M, \mathsf{cmd}(\mathsf{args})) = (\perp, \mathrm{u}, M)$*, in all the other cases.*

Given a sequence of simulation scenarios (formally represented as disturbance traces), we can build a sequence of commands (*simulation campaign*) driving the simulator through such scenarios. We define the simulator *output sequence* as the sequence of the SUV outputs associated with the simulator states traversed by a simulation campaign. Conversely, given a simulation campaign, we can compute the sequence of scenarios (*event lists*) simulated by it. These concepts are formalised in Definition 10.

Definition 10 (Simulation campaign and output sequence). *Let* $\mathcal{S} = (\mathcal{H}, L, W, m)$ *be a simulator and* $\mathsf{sim}_{\mathcal{S}}$ *be its transition function.*

(a) A simulation campaign *of length* c *for* $\mathcal{S}$ *is a sequence* $\chi = \mathsf{cmd}_0(\mathsf{args}_0), \ldots, \mathsf{cmd}_{c-1}(\mathsf{args}_{c-1})$ *of commands along with their arguments.*

(b) Each simulation campaign univocally defines a sequence of simulator states *traversed by the simulator while executing the simulation campaign. Formally, the* sequence of simulator states *of* $\mathcal{S}$ *with respect to a simulation campaign* χ *(as above) is* $(s_0, \mathrm{u}_0, M_0), \ldots, (s_c, \mathrm{u}_c, M_c)$ *where* s_0 *is the initial state of* $\mathcal{H}$*,* $\mathrm{u}_0 = \emptyset$*,* $M_0 = \emptyset$*, and for all* $0 \leq j < c$*,* $\mathsf{sim}_{\mathcal{S}}(s_j, \mathrm{u}_j, M_j, \mathsf{cmd}_j(\mathsf{args}_j)) = (s_{j+1}, \mathrm{u}_{j+1}, s_{j+1})$.

(c) A simulation campaign is admissible *if it is actually executable, i.e., iff* $s_c \neq \perp$.

(d) The output sequence *associated to an admissible simulation campaign* χ *is* $\mathit{output}(s_0), \mathit{output}(s_1), \ldots, \mathit{output}(s_c)$.

Note that, when $\mathcal{H}$ is a MDES (Definition 4), the output sequence of any simulation campaign χ on $\mathcal{S}$ is non-decreasing, as $s_0, \ldots, s_c$ are the S-components of the sequence of simulator states $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ traversed in that order. Therefore, the output sequence will be 0 as long as the property under verification is satisfied, and goes to (and stays at) 1 as soon as a property violation is detected by the monitor.

The event list sequence associated to a simulation campaign (Definition 11) is the sequence of the event lists associated to the simulator states where the simulator executes a *load* command, plus the event list associated to the simulator final state s_c. Forthcoming Definition 11 and Theorem 1 will be used to state (and prove) the correctness of our simulation campaign optimisation algorithm outlined in Section 5.

Definition 11 (Event list sequence associated to a simulation campaign). *Let $\mathcal{S} = (\mathcal{H}, L, W, m)$ be a simulator, $\mathsf{sim}_{\mathcal{S}}$ be its transition function, $\chi = \mathsf{cmd}_0(\mathsf{args}_0), \ldots, \mathsf{cmd}_{c-1}(\mathsf{args}_{c-1})$ a simulation campaign for $\mathcal{S}$ and $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ be the sequence of simulator states associated to χ.*

The event list sequence *associated to χ is $U(\chi) = \mathsf{u}_{j_0}, \ldots, \mathsf{u}_{j_{v-1}}, \mathsf{u}_c$, where, for all $0 \leq r < v$, u_{j_r} is the event list associated to the state where the simulator executes the r-th load command in χ (i.e., $\mathsf{cmd}_{j_r} = \mathsf{load}$ and there are exactly r load commands in χ before cmd_{j_r}), and u_c is the event list associated to the final simulator state.*

Theorem 1 links inputs (simulation campaigns) for a simulator $\mathcal{S}$ for $\mathcal{H}$ to inputs (event lists) for $\mathcal{H}$: for each simulation campaign χ, the event list u of any simulator state (s, u, M) traversed by $\mathcal{S}$ while executing χ drives $\mathcal{H}$ from its initial state s_0 to s.

Theorem 1. *Let $\mathcal{S} = (\mathcal{H}, L, W, m)$ be a simulator for $\mathcal{H}$, χ be an admissible simulation campaign for $\mathcal{S}$ of length c, and $(s_0, \mathsf{u}_0, M_0), \ldots, (s_c, \mathsf{u}_c, M_c)$ be the sequence of simulator states of $\mathcal{S}$ with respect to χ.*

For each $0 \leq j \leq c$, event list u_j in state (s_j, u_j, M_j) has form $(v_0, \tau_0), \ldots, (v_{q_j-1}, \tau_{q_j-1})$ and defines a discrete event sequence that drives $\mathcal{H}$ from its initial state s_0 to state s_j in time $T_j = \sum_{r=0}^{q_j-1} \tau_r$ (where $T_j = 0$ if $q_j = 0$).

4 Automatic Generation of Exhaustive Simulation Scenarios

In this section we outline our approach to disturbance modelling and disturbance trace generation. Each disturbance trace prefix identifies a simulator state. To allow generation and optimisation of simulation campaigns (Section 5), we associate a unique label to each of such prefixes (Definition 12).

Definition 12 (Labelling of disturbance traces). *Let $d \in \mathbb{N}^+$ and L be a countably infinite set of labels (e.g., $\mathbb{N}^+$). A labelling function over $[0, d]$ is an injective map λ from finite sequences of values in $[0, d]$ (including the empty sequence) to labels in L.*

Let $\delta = d_0, \ldots, d_{h-1}$ be a disturbance trace. The labelling of δ (according to λ) is $\delta' = l_0, d_0, \ldots, l_{h-1}, d_{h-1}, l_h$ where, for all $0 \leq i \leq h$, $l_i = \lambda(d_0, \ldots, d_{i-1})$.

We now outline our disturbance trace generation algorithm. We model the finite state automaton DG $\mathcal{D} = (Z, d, \mathit{dist}, \mathit{adm}, Z_I, Z_F)$ using the CMurphi [10,9] finite state

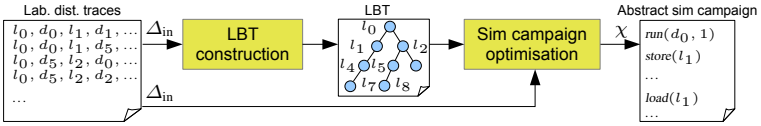


Fig. 2. High-level view of our simulation campaign optimiser

model checker, modified so as to generate all paths of length h from a DG initial state to a final state. Of course, any other finite state model checker, e.g., SPIN [14], could be used for this purpose. CMurphi has a rule-based modelling language: we define a disturbance with a rule whose *guard* and *body* define, respectively, functions *adm* and *dist* in $\mathcal{D}$. The labelling function is realised with a counter incremented each time a rule is fired (i.e., a disturbance is injected).

In order to enable a parallel approach to simulation, we partition the generated sequence of disturbance traces into $k \in \mathbb{N}^+$ subsequences (slices) of equal size. We will see in Section 5 that keeping together disturbance traces having a long common prefix enables better optimisation of simulation campaigns. This suggests to keep together traces generated consecutively by DFS. Thus, we first generate all n labelled disturbance traces into a single file and then we split such a file into k *slices* $\Delta^0_{\text{slice}}, \ldots, \Delta^{k-1}_{\text{slice}}$, by assigning the i-th trace ($0 \leq i < n$) to the $\left\lfloor \frac{ik}{n} \right\rfloor$-th slice.

5 Computation of Optimised Simulation Campaigns

Given a DG $\mathcal{D}$ and a sequence $\Delta_{\text{in}} = \delta_0, \ldots, \delta_{n-1}$ of labelled disturbance traces for $\mathcal{D}$, our simulation campaign optimiser (Fig. 2) computes a simulation campaign χ for *any* simulator $\mathcal{S}$ of *any* DES $\mathcal{H} = (S, s_0, d, O, \mathit{flow}, \mathit{jump}, \mathit{output})$ whose set of inputs $[0, d]$ is equal to the set of outputs of $\mathcal{D}$. The computed χ is *abstract* in that, for all commands of the form $\mathit{run}(e, t)$, t is a natural number and not an actual time duration. By providing a time step $\tau \in \mathbb{R}^+$, χ can be instantiated into a *concrete* simulation campaign χ_τ, by replacing all $\mathit{run}(e, t)$ commands by $\mathit{run}(e, t\tau)$.

The sequence of event lists $U(\chi_\tau)$ of χ_τ is *equal* (Theorem 2) to the sequence of event lists $u_\tau(\delta_0), \ldots, u_\tau(\delta_{n-1})$ associated to $\delta_0, \ldots, \delta_{n-1}$ with respect to time step τ. This implies that if the disturbance traces for a SLFV problem $(\mathcal{H}, \mathcal{D}, \tau, h)$ are split into $k \in \mathbb{N}^+$ slices $\Delta^0_{\text{slice}}, \ldots, \Delta^{k-1}_{\text{slice}}$, k instances of our optimiser can be used to *independently* compute k simulation campaigns, one for each slice. These can then be *independently* executed on k simulators. As the SUV $\mathcal{H}$ is an MDES (Definition 4), the answer to $(\mathcal{H}, \mathcal{D}, \tau, h)$ is *FAIL* iff the output of at least one simulator becomes 1 (Theorem 3). In that case, a *counterexample* can be derived from that simulator (Fig. 3).

We now outline our simulation campaign optimiser (Fig. 2). As the input sequence Δ_{in} of labelled disturbance traces can be too big to be kept in main memory, the optimiser reads the input file sequentially twice. In the first scan of Δ_{in}, the optimiser builds a data structure called Labels Branching Tree (LBT) as completely as possible within the available RAM. Afterwards, it reads Δ_{in} again to produce the abstract simulation campaign from the LBT, ensuring that the number of states stored on simulator side (by means of *store* and *free* commands) is always within the simulator capabilities. RAM and simulator memory management follows precise *policies* discussed next.

LBT Construction. The LBT is a tree of labels rooted at l_0, the first label of all traces. The LBT collects *branching labels*, i.e., labels l_i for which there exist at least two disturbance traces $\delta = l_0, d_0, \ldots, l_i, d_i, \ldots, l_h$ and $\delta' = l_0, d_0, \ldots, l_i, d'_i, \ldots, l'_h$ in Δ_{in} which are identical up to l_i and such that $d_i \neq d'_i$. Branching labels represent simulator states whose storing may save simulation time (by loading them back later).

Label l_j is a child of l_i in the LBT iff, for all $\delta = l_0, d_0, \ldots, l_i, \ldots, l_j, \ldots, l_h \in \Delta_{\text{in}}$, no l_k in δ with $i < k < j$ is in the LBT (note: all such δ are identical at least up to l_j). For each label in the LBT, the number of the first and last trace in Δ_{in} where it occurs are kept. To recognise branching labels, the optimiser needs to maintain in RAM already seen labels not yet proven to be branching. Whenever the optimiser runs short of memory, it forgets some of these labels. As this would prevent or delay the recognition of further branching labels (leading to a smaller LBT and causing the computation of a less optimised simulation campaign), the optimiser first forgets the deepest labels (less likely to become branching later).

Computation of the Abstract Simulation Campaign. Once the LBT is built, the optimiser reads Δ_{in} a second time to compute the abstract simulation campaign χ, keeping track of which LBT labels are stored in simulator memory at any moment.

For each $\delta = l_0, d_0, \ldots, l_{\text{load}}, \ldots, l_h$ in Δ_{in}, let l_{load} be the right-most label in the LBT currently stored by the simulator. The optimiser appends to χ the following commands: (i) *load*(l_{load}); (ii) a command of the form *run*$(\hat{d}, t)$ for each maximal subsequence of length t in δ (starting from l_{load}) of the form $\hat{d}, l_{i_1}, 0, l_{i_2}, \ldots, 0, l_{i_t}\tilde{d}, \tilde{l}$ where either $\tilde{d} \neq 0$ or label $\tilde{l}$ needs to be stored. In the latter case, command *store*$(\tilde{l})$ is appended as well. Label $\tilde{l}$ needs to be stored if it is in the LBT, it will occur again in a later trace, and simulator memory is not full. If the latter requirement fails, the optimiser first needs to free-up simulator memory: it selects a label l_{free} to free among all those already stored and $\tilde{l}$ itself, and appends command *free*(l_{free}) to χ. Label l_{free} is chosen among those that will not occur in later traces. If none exists, then l_{free} is chosen as to minimise the simulation cost (number of steps) to drive the simulator to the state represented by l_{free}, starting from the state represented by its parent label in the LBT.

Theorem 2 shows that commands in the abstract simulation campaign χ computed by the optimiser on input Δ_{in} drive $\mathcal{S}$ as to correctly simulate the effects of Δ_{in} on $\mathcal{H}$.

Theorem 2. *Let $(\mathcal{H}, \mathcal{D}, \tau, h)$ be a SLFV problem, $\mathcal{S} = (\mathcal{H}, L, W, m)$ a simulator for $\mathcal{H}$, and $\Delta_{in} = \delta_0, \ldots, \delta_{n-1}$ be an ordered sequence of* some *labelled disturbance traces for $\mathcal{D}$, each of which being of the form $\delta_i = l_0, d_0, \ldots, l_{h-1}, d_{h_i-1}, l_{h_i}$ ($0 \leq i < n$). Let $U_\tau(\Delta_{in})$ be the sequence $\mathsf{u}_\tau(\delta_0), \ldots, \mathsf{u}_\tau(\delta_{n-1})$ of event lists associated to the disturbance traces in Δ_{in} with respect to time step τ. The simulation campaign χ produced by the optimiser on input Δ_{in} is such that $U(\chi_\tau) = U_\tau(\Delta_{in})$ where χ_τ is the instantiation of χ with time step τ.*

Theorem 3 shows that if there exist disturbance traces in $\Delta^h_{\mathcal{D}}$ falsifying the property under verification, at least one of the generated campaigns returns a counterexample.

Theorem 3. *Let $(\mathcal{H}, \mathcal{D}, \tau, h)$ be a SLFV problem, $k \in \mathbb{N}^+$, $\mathcal{S}^0, \ldots, \mathcal{S}^{k-1}$ be k simulators for $\mathcal{H}$, and $\Delta^h_{\mathcal{D}}$ be a labelling of all disturbance traces for $\mathcal{D}$ of length h.*

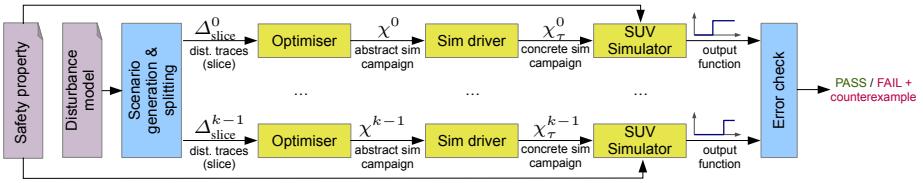


Fig. 3. Our overall approach

Let $\Delta^0_{slice}, \ldots, \Delta^{k-1}_{slice}$ *be a partition of* $\Delta^h_{\mathcal{D}}$ *into* k *sequences. Let* $\chi^0_\tau, \ldots, \chi^{k-1}_\tau$ *be the instantiations with time step* τ *of the abstract simulation campaigns* $\chi^0, \ldots, \chi^{k-1}$ *computed by the optimiser on inputs* $\Delta^0_{slice}, \ldots, \Delta^{k-1}_{slice}$.

The answer to $(\mathcal{H}, \mathcal{D}, \tau, h)$ *is FAIL iff there exists* $0 \leq j < k$ *such that the state sequence of simulator* $\mathcal{S}^j$ *on* χ^j_τ *contains a state* (s^*, u^*, M^*) *such that output*$(s^*) = 1$.

6 Experimental Results

In this section we present experimental results in order to evaluate the effectiveness of our SLFV approach summarised in Fig. 3.

From Fig. 3 we see that the disturbance model and thus disturbance generation and simulation campaign optimisation do not depend on the SUV model, which only affects the simulation time. For this reason we experiment with just one *large* SUV model. As our optimiser (Section 5) takes as input a set of disturbance traces (Fig. 3), disturbance models generating the same set of disturbance traces will yield the same results. For this reason we only focus on one disturbance model and change the number of the disturbance traces given as input to our optimiser in order to evaluate its performance.

Our optimiser computes an *abstract* simulation campaign χ. In order to execute χ, we need a *driver* that instantiates χ with a time step τ into χ_τ and translates χ_τ into commands for the target simulator. We implemented such a driver (*Sim driver* in Fig. 3) for the Simulink simulator and performed SLFV of the fuel control system in the Simulink distribution. This example has been studied in [31] using statistical model checking techniques. The fuel control system has three sensors subject to faults (disturbances). We verify one of the system level specifications for such a model, namely: the *fuel_air* model variable is never 0 for more than one second. Accordingly, our SUV consists of the Simulink model for the system along with a monitor for the property under verification. In our disturbance model, system sensors are subject to temporary faults, which are repaired after one second. In our setting, the complexity of the computation of an optimised simulation campaign does not depend on the SUV, it primarily depends on the number of disturbance traces to be simulated. Thus, the worst case for our approach is when all disturbance traces have to be simulated, i.e., when the answer to the SLFV problem is *PASS*. We know that this is the case when no more than one fault occurs within a second, thus this will be our disturbance model.

Experiments are performed on multiple 3.0 GHz, 8GB RAM Intel hyperthreaded Quad Core Linux PCs. Our time step τ (quantum between disturbances) is 1 second.

Table 1. Experimental results

h	time (h:m:s)	#traces	file size (MB)
50	0:1:35	448,105	195.725
60	0:3:29	805,075	420.743
70	0:6:35	1,314,145	799.584
80	0:11:41	2,002,315	1,390.157
90	0:21:34	2,896,585	2,259.642
100	0:28:39	4,023,955	3,484.489

(a) Disturbance trace generation

k	time (h:m:s)	slice size (MB)
2	0:0:14	1,742.244
4	0:0:14	871.122
8	0:0:15	435.561
16	0:0:14	217.78
32	0:0:14	108.89
64	0:0:13	54.445

(b) Instance $h = 100$ splitting

k	#traces	LBT size	$m = 1$ time	$m = 1$ #cmds	$m = 100{,}000$ time	$m = 100{,}000$ #cmds	%opt
2	2,011,977	670,661	0:3:14	16,040,520	3:47:57	8,047,912	**79.42%**
4	1,005,988	335,331	0:2:28	8,012,662	1:45:04	4,023,955	**83.32%**
8	502,994	167,666	0:0:35	4,001,378	0:44:27	2,011,978	**86.49%**
16	251,497	83,834	0:0:18	1,997,486	0:16:24	1,005,991	**88.97%**
32	125,748	41,918	0:0:07	996,660	0:4:50	502,996	**90.87%**
64	62,874	20,959	0:0:03	496,906	0:0:51	251,497	**92.47%**

(c) Simulation campaign optimisation ($h = 100$, time in h:m:s)

k	$m = 1$ time	$m = 100{,}000$ time	speedup
8	n/a	29, 13:50:12	$> \mathbf{1.7}\times$
16	n/a	14, 6:39:09	$> \mathbf{3.5}\times$
32	25, 23:07:43	6, 22:32:25	$\mathbf{3.8}\times$
64	12, 22:58:16	3, 9:19:18	$\mathbf{3.8}\times$

(d) Simulation (time in days, h:m:s)
'n/a' Simulation aborted after 50 days

k	offline gener.	offline split.	offline optimis.	offline total	online simulation	%offline	%online
8	0:28:39	0:0:15	0:44:27	1:13:21	29, 13:50:12	**0.17%**	99.83%
16	0:28:39	0:0:14	0:16:24	0:45:17	14, 6:39:09	**0.22%**	99.78%
32	0:28:39	0:0:14	0:4:50	0:33:43	6, 22:32:25	**0.34%**	99.66%
64	0:28:39	0:0:13	0:0:51	0:29:43	3, 9:19:18	**0.31%**	99.69%

(e) Offline vs. online phase (time in days, h:m:s)

Automatic Generation of Exhaustive Simulation Scenarios. Table 1a shows the time needed by our disturbance generator (Section 4) to generate disturbance traces with different horizons (column h). Our experiments show that the generation of even millions of traces is done in a matter of minutes. In the following, we focus on experiment $h = 100$ in Table 1a, since it has the largest sequence of disturbance traces (4,023,955).

Table 1b shows the time needed to split (Section 4) the sequence of disturbance traces of experiment $h = 100$ in Table 1a to enable parallel computation of simulation campaigns, with different degrees of parallelism (column k). Our experiments show that such tasks take just a few seconds.

Computation of Optimised Simulation Campaigns. Table 1c shows the performance of our optimiser (Section 5) when computing a simulation campaign from a slice of disturbance trace sequence (one k-th of the disturbance traces of instance $h = 100$). The table shows the number of traces of each slice, the size of the LBT, the time to compute the simulation campaign as well as the number of commands it consists of in two scenarios: columns $m = 1$ refer to computations of *unoptimised* simulation campaigns (as only the initial state can be stored in the simulator), while those for $m =$100,000 refer to *optimised* campaigns for a simulator with about 15GB of disk

space available (as, in our case study, each stored state takes about 150KB). Column *%opt* shows how much the simulation campaigns for m =100,000 are optimised with respect to the case with $m = 1$. Namely, *%opt* is the average value of L_l/h, where L_l is the number (at most h) of simulation time steps we save thanks to command *load*(l).

Execution of the Simulation Campaigns. Table 1d shows the time needed to execute our simulation campaigns on Simulink. For each row (degree of parallelism k) we report the maximum of the time needed by Simulink to execute the k simulation campaigns forming the verification task for $m = 1$ and $m = 100{,}000$.

The parallelism enabled by our approach is essential to handle large simulation campaigns as those considered here. In fact, for $k < 8$, we could not complete the simulation in 50 days (while we can easily compute the simulation campaign, Table 1c).

Summing Up. Table 1e sums up our results by showing the total time spent offline computing the optimised simulation campaign (column *total*), the time spent online by executing the simulation campaign (column *online simulation*) and the (percentage of the) time spent in the offline [online] computation (column *%offline*) [(column *%online*)]. We can see that our offline computations account for less than 0.5% of the total simulation time and, most importantly, enable exhaustive parallel HILS almost 4 times faster than without optimisation (Table 1d).

7 Conclusions

We have presented a HILS based approach to SLFV. We use explicit model checking techniques to model, generate and optimise exhaustive simulation scenarios for parallel HILS. This enables *black box* SLFV of *actual* systems. We have shown the effectiveness of our approach by applying it to a large control system case study in the Simulink distribution. To the best of our knowledge, this is the first time that exhaustive HILS has been carried out on a set of simulation scenarios (disturbance traces) of the size considered here (about 4 millions). Our experimental results show that we spend more than 99% of the SLFV time in the simulation activity. Thus, investigation of guided search techniques, for example as in [7], is a promising future work in our setting.

Acknowledgements. Work partially supported by FP7 projects SmartHG (317761) and PAEON (600773). We thank our reviewers for their valuable comments to our paper.

References

1. Alur, R.: Formal verification of hybrid systems. In: Proc. EMSOFT 2011. ACM (2011)
2. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DIVINE – A tool for distributed verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
3. Bingham, B., Bingham, J., De Paula, F.M., Erickson, J., Singh, G., Reitblatt, M.: Industrial strength distributed explicit state model checking. In: Proc. PDMC-HIBI 2010. IEEE (2010)

4. Brillout, A., He, N., Mazzucchi, M., Kroening, D., Purandare, M., Rümmer, P., Weissenbacher, G.: Mutation-based test case generation for simulink models. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 208–227. Springer, Heidelberg (2010)
5. Broy, M., Jonsson, B., Katoen, J.-P., Leucker, M., Pretschner, A. (eds.): Model-Based Testing of Reactive Systems. LNCS, vol. 3472. Springer, Heidelberg (2005)
6. Cavaliere, F., Mari, F., Melatti, I., Minei, G., Salvo, I., Tronci, E., Verzino, G., Yushtein, Y.: Model checking satellite operational procedures. In: Proc. DASIA 2011 (2011)
7. De Paula, F.M., Hu, A.J.: An effective guidance strategy for abstraction-guided simulation. In: Proc. DAC 2007, pp. 63–68. ACM (2007)
8. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. In: Proc. OSDI 2004. USENIX Association (2004)
9. Della Penna, G., Intrigila, B., Melatti, I., Tronci, E., Zilli, M.: Exploiting transition locality in automatic verification of finite state concurrent systems. STTT 6(4), 320–341 (2004)
10. Dill, D.L., Drexler, A.J., Hu, A.J., Han Yang, C.: Protocol verification as a hardware design aid. In: Proc. IEEE Int. Conf. Comp. Design on VLSI in Comp. & Proc., 1991. IEEE (1992)
11. Gadkari, A.A., Yeolekar, A., Suresh, J., Ramesh, S., Mohalik, S., Shashidhar, K.C.: AutoMOTGen: Automatic model oriented test generator for embedded control systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 204–208. Springer, Heidelberg (2008)
12. Grosu, R., Smolka, S.A.: Monte carlo model checking. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 271–286. Springer, Heidelberg (2005)
13. Ho, P.H., Shiple, T., Harer, K., Kukula, J., Damiano, R., Bertacco, V., Taylor, J., Long, J.: Smart simulation using collaborative formal and simulation engines. In: Proc. ICCAD 2000 (2000)
14. Holzmann, G.J.: The SPIN model checker. Addison-Wesley (2003)
15. Holzmann, G.J.: Parallelizing the spin model checker. In: Donaldson, A., Parker, D. (eds.) SPIN 2012. LNCS, vol. 7385, pp. 155–171. Springer, Heidelberg (2012)
16. Holzmann, G.J., Joshi, R., Groce, A.: Model driven code checking. Autom. Softw. Eng. 15(3-4), 283–297 (2008)
17. Kanade, A., Alur, R., Ivančić, F., Ramesh, S., Sankaranarayanan, S., Shashidhar, K.C.: Generating and analyzing symbolic traces of simulink/Stateflow models. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 430–445. Springer, Heidelberg (2009)
18. Maler, O., Nickovic, D.: Monitoring temporal properties of continuous signals. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 152–166. Springer, Heidelberg (2004)
19. Meenakshi, B., Bhatnagar, A., Roy, S.: Tool for translating simulink models into input language of a model checker. In: Liu, Z., Kleinberg, R.D. (eds.) ICFEM 2006. LNCS, vol. 4260, pp. 606–620. Springer, Heidelberg (2006)
20. Melatti, I., Palmer, R., Sawaya, G., Yang, Y., Kirby, R.M., Gopalakrishnan, G.: Parallel and distributed model checking in eddy. Int. J. Softw. Tools Technol. Transf. 11(1), 13–25 (2009)
21. Nanshi, K., Somenzi, F.: Guiding simulation with increasingly refined abstract traces. In: Proc. DAC 2006, pp. 737–742. ACM (2006)
22. Rozier, K.Y., Vardi, M.Y.: Deterministic compilation of temporal safety properties in explicit state model checking. In: Proc. HVC 2012. Springer (2012)
23. Sen, K., Viswanathan, M., Agha, G.: On statistical model checking of stochastic systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 266–280. Springer, Heidelberg (2005)
24. Sontag, E.D.: Mathematical Control Theory: Deterministic Finite Dimensional Systems. Texts in Applied Mathematics. Springer (1998)
25. Stern, U., Dill, D.L.: Parallelizing the Murphi Verifier. Form. Methods Syst. Des. 18(2), 117–129 (2001)

26. Tripakis, S., Sofronis, C., Caspi, P., Curic, A.: Translating discrete-time Simulink to Lustre. ACM Trans. Emb. Comp. Syst. 4(4), 779–818 (2005)
27. Tronci, E., Della Penna, G., Intrigila, B., Zilli, M.: A probabilistic approach to automatic verification of concurrent systems. In: Proc. APSEC 2001, pp. 317–324. IEEE (2001)
28. Venkatesh, R., Shrotri, U., Darke, P., Bokil, P.: Test generation for large automotive models. In: Proc. ICIT 2012, pp. 662–667. IEEE (2012)
29. Whalen, M., Cofer, D., Miller, S., Krogh, B.H., Storm, W.: Integration of formal analysis into a model-based software development process. In: Leue, S., Merino, P. (eds.) FMICS 2007. LNCS, vol. 4916, pp. 68–84. Springer, Heidelberg (2008)
30. Yang, C.H., Dill, D.L.: Validation with guided search of the state space. In: Proc. DAC 1998, pp. 599–604. ACM (1998)
31. Zuliani, P., Platzer, A., Clarke, E.M.: Bayesian statistical model checking with application to simulink/stateflow verification. In: Proc. HSCC 2010, pp. 243–252 (2010)

Beautiful Interpolants

Aws Albarghouthi[1] and Kenneth L. McMillan[2]

[1] University of Toronto
[2] Microsoft Research

Abstract. We describe a compositional approach to Craig interpolation based on the heuristic that simpler proofs of special cases are more likely to generalize. The method produces simple interpolants because it is able to summarize a large set of cases using one relatively simple fact. In particular, we present a method for finding such simple facts in the theory of linear rational arithmetic. This makes it possible to use interpolation to discover inductive invariants for numerical programs that are challenging for existing techniques. We show that in some cases, the compositional approach can also be more efficient than traditional lazy SMT as a decision procedure.

1 Introduction

beau·ti·ful *adjective* \'byü-ti-fəl\
1: *...exciting aesthetic pleasure*
2: *generally pleasing* [2]

In mathematics and physics, the beauty of a theory is an important quality. A simple or elegant argument is considered more likely to generalize than a complex one. Imagine, for example, proving a conjecture about an object in N dimensions. We might first try to prove the special case of two or three dimensions, and then generalize the argument to the N-dimensional case. We would prefer a proof of the two-dimensional case that is simple, on the grounds that it will be less prone to depend on particular aspects of this case, thus more likely to generalize.

We can apply this heuristic to the proof of programs or hardware systems. We produce a proof of correctness for some bounded collection of program executions. From this proof we can derive a conjectured invariant of the program using Craig interpolation methods, e.g., [22,24,21,11]. The simpler our conjecture, the less it is able to encode particular aspects of our bounded behaviors, so the more likely it is to be inductive. Typically, our bounded proofs will be produced by an SMT (satisfiability modulo theories) solver. Simplicity of our interpolant-derived conjecture depends on simplicity of the SMT solver's proof. Unfortunately, for reasons we will discuss shortly, SMT solvers may produce proofs that are far more complex than necessary.

In this paper, we consider an approach we call *compositional SMT* that is geared to produce simple interpolants. It is compositional in the sense that the interpolant acts as an intermediate assertion between components of the formula, localizing the reasoning. This approach allows us to solve inductive invariant

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 313–329, 2013.

generation problems that are difficult for other techniques, and in some cases can solve bounded verification problems much more efficiently than standard lazy SMT [8] methods. The approach is simple to implement and uses an unmodified SMT solver as a "black box".

A lazy SMT solver separates the problems of Boolean and theory reasoning. To test satisfiability of a formula A relative to a theory $\mathcal{T}$, it uses a SAT solver to find propositionally satisfying assignments. These can be thought of as disjuncts in the disjunctive normal form (DNF) of A. A theory solver then determines feasibility of these disjuncts in $\mathcal{T}$. In the negative case, it produces a *theory lemma*. This is a validity of the theory that contradicts the disjunct propositionally. In the worst case, each theory lemma rules out only one amongst an exponential collection of disjuncts.

In compositional SMT, we refute satisfiability of a conjunction $A \wedge B$ by finding an *interpolant* formula I, such that $A \Rightarrow I$, $B \Rightarrow \neg I$ and I uses only the symbols common to A and B. We do this by building two collections of feasible disjuncts of A and B that we call *samples*. We then try to construct a simple interpolant I for the two sample sets. If I is an interpolant for A and B, we are done. Otherwise, we use our SMT solver to find a new sample that contradicts either $A \Rightarrow I$ or $B \Rightarrow \neg I$, and restart the process with the new sample.

Unlike the theory solver in lazy SMT, our interpolant generator can "see" many different cases and try to find a simple fact that generalizes them. This more global view allows compositional SMT to find very simple interpolants in cases when lazy SMT produces an exponential number of theory lemmas.

We develop the technique here in the quantifier-free theory of linear rational arithmetic (QFLRA). This allows us to apply some established techniques based on Farkas' lemma to search for simple interpolants. The contributions of this paper are (1) A compositional approach to SMT based on sampling and interpolation (2) An interpolation algorithm for QFLRA based on finding linear separators for sets of convex polytopes. (3) A prototype implementation that demonstrates the utility of the technique for invariant generation, and shows the potential to speed up SMT solving.

Organization. Sec. 2 illustrates our approach on a simple example. Sec. 3 gives a general algorithm for interpolation via sampling. Sec. 4 describes an interpolation technique for sets of convex polytopes. Sec. 5 presents our implementation and experimental results. Related work is discussed in Sec. 6.

2 Motivating Example

Figure 1 shows two QFLRA formulas A and B over the variables x and y. For clarity of presentation, the two formulas are in DNF, where $\mathsf{A}_1, \mathsf{A}_2$, and A_3 are the disjuncts of A, and B_1 and B_2 are the disjuncts of B. Intuitively, since a disjunct is a conjunction of linear inequalities (*half-spaces*), it represents a *convex polyhedron* in $\mathbb{R}^2$. Fig. 2(a) represents A and B geometrically, where each disjunct is highlighted using a different shade of gray.

$$\begin{aligned}
\mathsf{A} = &(x \leq 1 \wedge y \leq 3) && (\mathsf{A}_1) \\
&\vee (1 \leq x \leq 2 \wedge y \leq 2) && (\mathsf{A}_2) \\
&\vee (2 \leq x \leq 3 \wedge y \leq 1) && (\mathsf{A}_3)
\end{aligned}
\qquad
\begin{aligned}
\mathsf{B} = &(x \geq 2 \wedge y \geq 3) && (\mathsf{B}_1) \\
&\vee (x \geq 3 \wedge 2 \leq y \leq 3) && (\mathsf{B}_2)
\end{aligned}$$

Fig. 1. Inconsistent formulas A and B

Since A and B do not intersect, as shown in Fig 2(a), $\mathsf{A} \wedge \mathsf{B}$ is unsatisfiable. Thus, there exists an interpolant I such that $\mathsf{A} \Rightarrow \mathsf{I}$ and $\mathsf{I} \Rightarrow \neg\mathsf{B}$. One such I is the half-space $x + y \leq 4$, shown in Fig. 2(b) as the region encompassing all of A, but not intersecting with B. We now discuss how our technique constructs such an interpolant.

We start by *sampling* disjuncts from A and B. In practice, we sample a disjunct from a formula φ by finding a model $m \models \varphi$, using an SMT solver, and evaluating all linear inequalities occurring in φ with respect to m. Suppose we sample the disjuncts A_2 and B_1. We now find an interpolant of $(\mathsf{A}_2, \mathsf{B}_1)$. To do so, we utilize Farkas' lemma and encode a system of constraints that is satisfiable *iff* there exists a *half-space interpolant* of $(\mathsf{A}_2, \mathsf{B}_1)$. That is, we are looking for an interpolant comprised of a single linear inequality, not an arbitrary Boolean combination of linear inequalities. In this case, we might find the half-space interpolant $y \leq 2.5$, shown in Fig. 2(c).

We call $y \leq 2.5$ a *partial interpolant*, since it is an interpolant of A_2 and B_1, which are parts of (i.e., subsumed by) A and B, respectively. We now check if this partial interpolant is an interpolant of (A, B) using an SMT solver. First, we check if $\mathsf{A} \wedge y > 2.5$ is satisfiable. Since it is satisfiable, $\mathsf{A} \not\Rightarrow y \leq 2.5$, indicating that $y \leq 2.5$ is not an interpolant of (A, B). A satisfying assignment of $\mathsf{A} \wedge y > 2.5$ is a model of A that lies outside the region $y \leq 2.5$, for example, the point $(1, 3)$ shown in Fig. 2(c). Since $(1, 3)$ is a model of the disjunct A_1, we add A_1 to the set of samples in order to take it into account.

At this point, we have two A samples, A_1 and A_2, and one B sample B_1. We now seek an interpolant for $(\mathsf{A}_1 \vee \mathsf{A}_2, \mathsf{B}_1)$. Of course, we can construct such an interpolant by taking the disjunction of two half-space interpolants: one for $(\mathsf{A}_1, \mathsf{B}_1)$ and one for $(\mathsf{A}_2, \mathsf{B}_1)$. Instead, we attempt to find a *single* half-space that is an interpolant of $(\mathsf{A}_1 \vee \mathsf{A}_2, \mathsf{B}_1)$ – we say that the samples A_1 and A_2 are *merged* into a *sampleset* $\{\mathsf{A}_1, \mathsf{A}_2\}$. As before, we construct a system of constraints and solve it for such an interpolant. In this case, we get the half-space $x + y \leq 4$, shown in Fig. 2(d). Since $x + y \leq 4$ is an interpolant of (A, B), the algorithm terminates successfully. If there is no half-space interpolant for $(\mathsf{A}_1 \vee \mathsf{A}_2, \mathsf{B}_1)$, we *split* the sampleset $\{\mathsf{A}_1, \mathsf{A}_2\}$ into two samples, and find two half-space interpolants for $(\mathsf{A}_1, \mathsf{B}_1)$ and $(\mathsf{A}_2, \mathsf{B}_1)$.

The key intuition underlying our approach is two-fold: (1) Lazily sampling a small number of disjuncts from A and B often suffices for finding an interpolant for all of A and B. (2) By merging samples, e.g., as A_1 and A_2 above, and encoding a system of constraints to find half-space interpolants, we are forcing

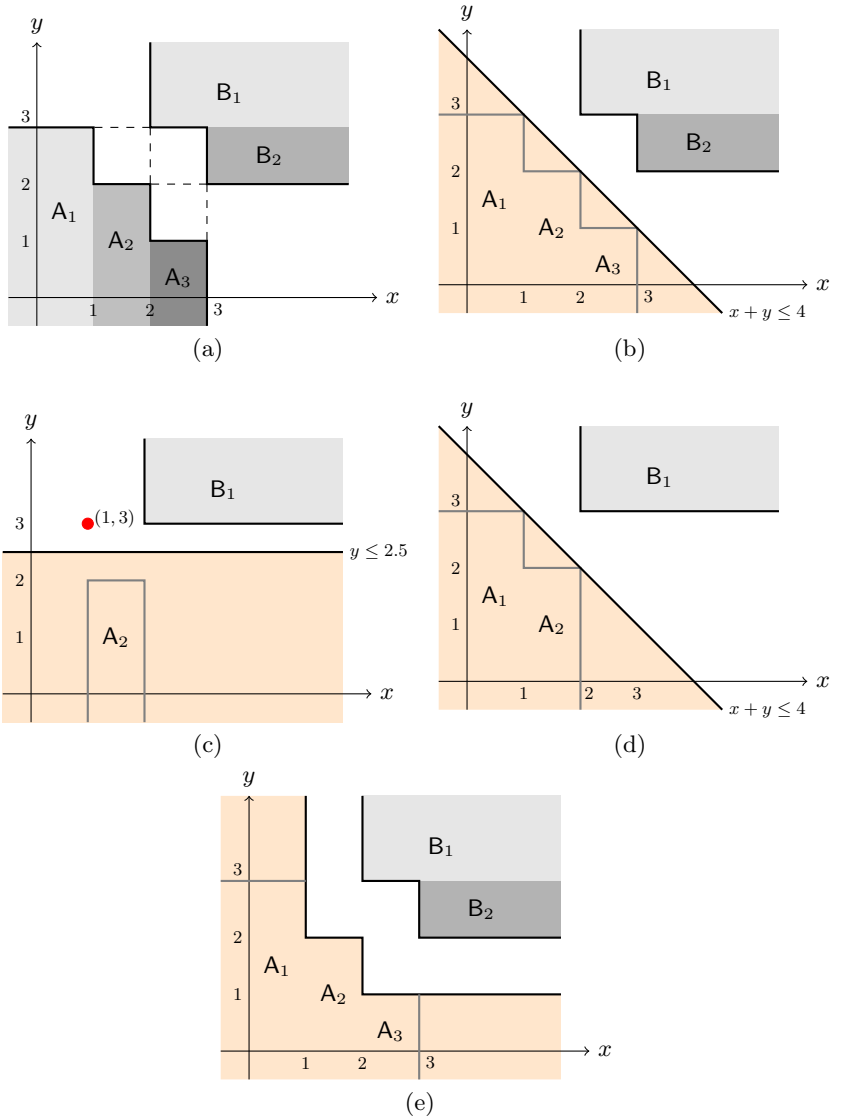


Fig. 2. (a) Illustration of the formulas A and B. (b) An interpolant $x + y \leq 4$ for (A, B). (c) An interpolant $y \leq 2.5$ for $(\mathsf{A}_2, \mathsf{B}_1)$. (d) An interpolant $x + y \leq 4$ for samples $(\mathsf{A}_1 \vee \mathsf{A}_2\, \mathsf{B}_1)$. (e) An interpolant computed by MathSAT5.

the procedure to take a holistic view of the problem and produce simpler and possibly more general interpolants.

Given the formulas A and B, an SMT solver is generally unable to find the simple fact $x + y \leq 4$, due to the specificity of the theory lemmas it produces.

For example, we used the MathSAT SMT solver to find an interpolant for A and B, and it produced the following formula[1]:

$$(x \leq 2 \wedge y \leq 2) \vee ((x \leq 1 \vee y \leq 1) \wedge ((x \leq 2 \wedge y \leq 2) \vee (x \leq 1 \vee y \leq 1))),$$

illustrated in Fig. 2(e). This interpolant is more complex and does not capture the emerging pattern of the samples. As we add more disjuncts to A and B following the pattern, the interpolants produced by MathSAT grow in size, whereas our approach produces the same result. This ability to generalize a series is key to invariant generation. In the SMT approach, the theory solver sees only one pair of disjuncts from A and B at a time, and thus cannot generalize.

3 Constructing Interpolants from Samples

We now present Csmt (Compositional SMT), a generic algorithm for computing an interpolant for a pair of quantifier-free first-order formulas (A, B). Csmt attempts to partition samples from A and B as coarsely as possible into *samplesets*, such that each A sampleset can be separated from each B sampleset by an atomic interpolant formula (for linear arithmetic, this means a single linear constraint). Although Csmt applies to any theory that allows quantifier-free interpolation, for concreteness, we consider here only linear arithmetic.

Preliminaries. A formula of *quantifier-free linear rational arithmetic*, LRA, is a Boolean combination of *atoms*. Each atom is a linear inequality of the form $c_1x_1 + \cdots + c_nx_n \lhd k$, where $c_1, \ldots, c_n$ and k are rational constants, $x_1, \ldots, x_n$ are distinct variables, and $\lhd$ is either $<$ or $\leq$. The atom is an *open* or *closed half-space* if $\lhd$ is $<$ or $\leq$, respectively. We use $\mathsf{Lin}\mathsf{Iq}(\varphi)$ to denote the set of atoms appearing in formula φ, and $\mathsf{Vars}(\varphi)$ to denote the set of variables. We will often write $cx \lhd k$ for a half-space $c_1x_1 + \cdots + c_nx_n \lhd k$, where c is the row vector of the coefficients $\{c_i\}$, and x is the column vector of the variables $\{x_i\}$.

A *model* of φ is an assignment of rational values to $\mathsf{Vars}(\varphi)$ that makes φ true. Given a model m of φ, we define the *sample* of φ w.r.t. m, written $sample_{\varphi}(m)$, as the formula

$$\bigwedge\{P \mid P \in \mathsf{Lin}\mathsf{Iq}(\varphi), m \models P\} \wedge \bigwedge\{\neg P \mid P \in \mathsf{Lin}\mathsf{Iq}(\varphi), m \not\models P\}.$$

Note, there are finitely many samples of φ and φ is equivalent to the disjunction of its samples. Geometrically, each sample can be thought of as a convex polytope.

Given two formulas $A, B \in \mathsf{LRA}$ such that $A \wedge B$ is unsatisfiable, an *interpolant* of (A, B) is a formula $I \in \mathsf{LRA}$ such that $\mathsf{Vars}(I) \subseteq \mathsf{Vars}(A) \cap \mathsf{Vars}(B)$, $A \Rightarrow I$, and $B \Rightarrow \neg I$. An interpolant exists for every inconsistent (A, B).

General Algorithm. We formalize Csmt in Fig. 3 as a set of guarded commands. In each, if the condition above the line is satisfied, the assignment below may be executed. The state of Csmt is defined by the following variables:

[1] Produced using MathSAT 5.2.2, and slightly modified for clarity of presentation.

$$\frac{}{\mathsf{S}_A := \emptyset \quad \mathsf{S}_B := \emptyset \quad \mathsf{PItp} := \emptyset}\ \textsc{Init}$$

• Rules for distributing samples into samplesets:

$$\frac{X \in \{A, B\} \quad s_1 \in \mathsf{S}_X \quad s_2 \in \mathsf{S}_X \quad s_1 \neq s_2}{\mathsf{S}_X := (\mathsf{S}_X \setminus \{s_1, s_2\}) \cup \{s_1 \cup s_2\}}\ \textsc{Merge}$$

$$\frac{X \in \{A, B\} \quad s_1 \cup s_2 \in \mathsf{S}_X \quad s_1 \neq \emptyset \quad s_2 \neq \emptyset}{\mathsf{S}_X := (\mathsf{S}_X \setminus \{s_1 \cup s_2\}) \cup \{s_1\} \cup \{s_2\}}\ \textsc{Split}$$

• Rules for constructing and checking interpolants:

$$\frac{s_A \in \mathsf{S}_A \quad s_B \in \mathsf{S}_B \quad \textsc{HalfItp}(s_A, s_B) \neq \diamond \quad \mathsf{PItp}(s_A, s_B) \text{ is undefined}}{\mathsf{PItp}(s_A, s_B) := \textsc{HalfItp}(s_A, s_B)}\ \textsc{PartialItp}$$

$$\frac{C = \textsc{Cand}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{PItp}) \neq \diamond \quad m \models A \wedge \neg C}{\mathsf{S}_A := \mathsf{S}_A \cup \{\{\mathit{sample}_A(m)\}\}}\ \textsc{CheckItpA}$$

$$\frac{C = \textsc{Cand}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{PItp}) \neq \diamond \quad m \models C \wedge B}{\mathsf{S}_B := \mathsf{S}_B \cup \{\{\mathit{sample}_B(m)\}\}}\ \textsc{CheckItpB}$$

• Termination conditions:

$$\frac{s_A \in \mathsf{S}_A \quad s_B \in \mathsf{S}_B \quad |s_A| = |s_B| = 1 \quad \textsc{HalfItp}(s_A, s_B) = \diamond}{A \wedge B \text{ is satisfiable}}\ \textsc{Sat}$$

$$\frac{C = \textsc{Cand}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{PItp}) \quad A \Rightarrow C \quad C \Rightarrow \neg B}{C \text{ is an interpolant of } (A, B)}\ \textsc{Unsat}$$

Fig. 3. CSMT as guarded commands

– *Sampleset collections* $\mathsf{S}_A, \mathsf{S}_B$ are sets of samplesets of A and B, respectively. Initially, as dictated by the command INIT, $\mathsf{S}_A = \mathsf{S}_B = \emptyset$.
– *Partial interpolant map* PItp is a map from pairs of samplesets to half-spaces. Invariantly, if (s_A, s_B) is in the domain of PItp then $\mathsf{PItp}(s_A, s_B)$ is an interpolant for $(\bigvee s_A, \bigvee s_B)$.

We do not attempt to find a smallest set of half-spaces that separate the samples, as this problem is NP-complete. This can be shown by reduction from *k-polyhedral separability*: given two sets of points on a plane, is there a set of less than k half-spaces separating the two sets of points [26]. Instead, we heuristically cluster the samples into large samplesets such that each pair of samplesets (s_A,s_B) is linearly separable. Even with a minimal clustering, this solution may be sub-optimal, in the sense of using more half-spaces than necessary. Since our objective is a heuristic one, we will seek reasonably simple interpolants with moderate effort, rather than trying to optimize.

In practice, we heuristically search the space of clusterings using Merge and Split. Merge is used to combine two samplesets in $\mathsf{S}_{\{A,B\}}$ and make them a single sampleset. Split performs the opposite of Merge: it picks a sampleset in $\mathsf{S}_{\{A,B\}}$ and splits it into two samplesets. The command PartialItp populates the map Pltp with interpolants for pairs of samplesets (s_A, s_B). This is done by calling HalfItp (s_A, s_B), which returns a half-space interpolant of $(\bigvee s_A, \bigvee s_B)$ if one exists, and the symbol $\diamond$ otherwise (Sec. 4 presents an implementation of HalfItp).

The commands CheckItpA and CheckItpB check if the current *candidate interpolant* is *not* an interpolant of (A, B), in which case, samples produced from counterexamples are added to S_A and S_B. The function Cand constructs a candidate interpolant from Pltp. If the domain of Pltp contains $\mathsf{S}_A \times \mathsf{S}_B$, then

$$\textsc{Cand}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{Pltp}) \equiv \bigvee_{s_A \in \mathsf{S}_A} \left(\bigwedge_{s_B \in \mathsf{S}_B} \mathsf{Pltp}(s_A, s_B) \right)$$

Otherwise the result is $\diamond$. The result of Cand is an interpolant of (A', B'), where A' and B' are the disjunction of all samples in S_A and S_B, respectively. This DNF formula may not be optimal in size. Synthesizing optimal candidate interpolants from partial interpolants is a problem that we hope to explore in the future.

If Sat or Unsat apply, then the algorithm terminates. Sat checks if two singleton samplesets do not have a half-space separating them. Since both samples define convex polytopes, if no half-space separates them, then they intersect, and therefore $A \wedge B$ is satisfiable. Unsat checks if a candidate interpolant C is indeed an interpolant, in which case Csmt terminates successfully.

Example 1. Consider the formulas A and B from Sec. 2. Suppose that Csmt is in the state $\mathsf{S}_A = \{\{\mathsf{A}_2\}\}$, $\mathsf{S}_B = \{\{\mathsf{B}_1\}\}$, $\mathsf{Pltp} = \emptyset$, where A_2 and B_1 are the samples of A and B defined in Sec. 2. By applying PartialItp, we find a half-space separating the only two samplesets. As a result, $\mathsf{Pltp}(\{\mathsf{A}_2\}, \{\mathsf{B}_1\}) = y \leq 2.5$. Suppose we now apply CheckItpA. The candidate interpolant $\textsc{Cand}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{Pltp})$ at this point is $y \leq 2.5$. Since $\mathsf{A} \wedge y > 2.5$ is satisfiable, CheckItpA adds the sampleset $\{\mathsf{A}_1\}$, which is not subsumed by the candidate interpolant, to S_A. Now, $\mathsf{S}_A = \{\{\mathsf{A}_1\}, \{\mathsf{A}_2\}\}$. Since we have two samplesets in S_A, we apply Merge and get $\mathsf{S}_A = \{\{\mathsf{A}_1, \mathsf{A}_2\}\}$. PartialItp is now used to find a half-space interpolant for the samplesets $\{\mathsf{A}_1, \mathsf{A}_2\}$ and $\{\mathsf{B}_1\}$. Suppose it finds the plane $x + y \leq 4$. Then Unsat is applicable and the algorithm terminates with $x + y \leq 4$ as an interpolant of (A, B). □

The key rule for producing simpler interpolants is Merge, since it decreases the number of samplesets, and forces the algorithm to find a smaller number of half-spaces that separate a larger number of samples. In Example 1 above, if we do not apply Merge, we might end up adding all the samples of A and B to S_A and S_B. Thus, producing an interpolant with a large number of half-spaces like the one illustrated in Fig. 2(e).

Theorem 1 (Soundness of Csmt). *Given two formulas A and B, if* Csmt *terminates using*

1. SAT, *then* $A \wedge B$ *is satisfiable.*
2. UNSAT, *then* $\text{CAND}(\mathsf{S}_A, \mathsf{S}_B, \mathsf{PItp})$ *is an interpolant of* (A, B).

Proof (Sketch). In case 1, by definition of the rule SAT, we know that there is a sample a of A and a sample b of B such that there does not exist a half-space I that is an interpolant of (a, b). Since both a and b define convex polytopes, if a and b do not have a half-space separating them, then $a \wedge b$ is satisfiable, and therefore $A \wedge B$ is satisfiable.

In case 2, the candidate interpolant C, checked in rule UNSAT, is over the shared variables of A and B (by definition of HALFITP), $A \Rightarrow C$, and $C \Rightarrow \neg B$. Therefore, it is an interpolant of (A, B). □

We now consider the termination (completeness) of CSMT. It is easy to see that one can keep applying the commands MERGE and SPLIT without ever terminating. To make sure that does not happen, we impose the restriction that for any sampleset in $P_{\{A,B\}}$, if it is ever split, it never reappears in the sampleset collection. For example, if a sampleset $s_A \in \mathsf{S}_A$ is split into two samplesets s_A^1 and s_A^2 using SPLIT, then s_A cannot reappear in S_A. Given this restriction, CSMT always terminates.

Theorem 2 (Completeness of CSMT). *For any two formulas A and B,* CSMT *terminates.*

Proof (Sketch). Since there are finitely many samples, CHECKITPA, CHECKITPA and PARTIALITP must be executed finitely. Due to our restriction, MERGE is also bounded. Thus, if we do not terminate, eventually SPLIT reduces all samplesets to singletons, at which point SAT or UNSAT must terminate the procedure.

4 Half-Space Interpolants

In this section, we present a constraint-based implementation of the parameter HALFITP of CSMT. Given two samplesets s_A and s_B, our goal is to find a half-space interpolant $ix \triangleleft k$ of $(\bigvee s_A, \bigvee s_B)$. Since both s_A and s_B represent a union (set) of convex polytopes, we could compute the *convex hulls* of s_A and s_B and use techniques such as [29,7] to find a half-space separating the two convex hulls. To avoid the potentially expensive convex hull construction, we set up a system of linear constraints whose solution encodes both the separating half-space and the proof that it is separating. We then solve the constraints using an LP solver. This is an extension of the method in [29] from pairs of convex polytopes to pairs of *sets* of convex polytopes.

The intuition behind this construction is simple. We can represent the desired separator as a linear constraint I of the form $ix \leq k$, where x is a vector of variables, i is a vector of unknown coefficients, and k is an unknown constant. We wish to solve for the unknowns i and k. To express the fact that I is a separator, we apply Farkas' lemma. This tells us that a set of linear constraints

$\{C_j\}$ implies I exactly when I can be expressed as a linear combination of $\{C_j\}$ with non-negative coefficients. That is, when $\Sigma_j c_j C_j + d \equiv I$ for some non-negative $\{c_j\}$ and d. The key insight is that this equivalence is itself a set of linear equality constraints with unknowns $\{c_j\}$, d, i and k. The values of $\{c_j\}$ and d constitute a certificate that in fact $\{C_j\}$ implies I. We can therefore construct constraints requiring that each sample in s_A implies (is contained in) I, and similarly that each sample in s_B implies $\neg I$ (equivalent to $-ix < -k$). Solving these constraints we obtain a separator I and simultaneously a certificate that I is a separator. What is new here is only that we solve for multiple Farkas proofs: one for each sample in s_A or s_B.

We now make this construction precise. Let $N_A = |s_A|$ and $N_B = |s_B|$. Each sample in s_A is represented as a vector inequality $A_j x \leq a_j$, for $j \in [1, N_A]$. Similarly, samples in s_B are of the form $B_j x \leq b_j$, for $j \in [1, N_B]$. Here, x is a column vector of the variables $\mathsf{Vars}(A) \cup \mathsf{Vars}(B)$. For example, the sample $y \leq 1 \wedge z \leq 3$ is represented as follows:

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} \begin{pmatrix} y \\ z \end{pmatrix} \leq \begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

In the remainder of this section, we assume that all samples are conjunctions of closed half-spaces, i.e., non-strict inequalities. (In [6], we present a simple extension to our construction for handling open half-spaces.) It follows that if there exists a half-space interpolant for $(\bigvee s_A, \bigvee s_B)$, then there exists a *closed* half-space that is also an interpolant. Thus, our goal is to find a closed half-space $ix \leq k$ that satisfies the following two conditions:

$$\forall j \in [1, N_A] \cdot A_j x \leq a_j \Rightarrow ix \leq k \tag{1}$$

$$\forall j \in [1, N_B] \cdot B_j x \leq b_j \Rightarrow \neg ix \leq k \tag{2}$$

Condition (1) specifies that $ix \leq k$ subsumes all s_A samples. Condition (2) specifies that $ix \leq k$ does not intersect with any of the s_B samples. By forcing coefficients of unshared variables to be 0 in $ix \leq k$, we ensure that $ix \leq k$ is an interpolant of $(\bigvee s_A, \bigvee s_B)$. To construct such half-space interpolant, we utilize Farkas' lemma [30]:

Theorem 3 (Farkas' lemma). *Given a satisfiable system of linear inequalities $Ax \leq b$ and a linear inequality $ix \leq k$, then: $Ax \leq b \Rightarrow ix \leq k$ iff there exists a row vector $\lambda \geq 0$ s.t. $\lambda A = i$ and $\lambda b \leq k$.*

Using this fact, we construct a constraint Φ_A that, when satisfiable, implies that a half-space $ix \leq k$ satisfies condition (1). Consider a sample $A_j x \leq a_j$, where A_j is an $m_j \times n_j$ matrix. We associate with this sample a row vector λ_{A_j} of size m_j, consisting of fresh variables, called *Farkas coefficients* of the m_j linear inequalities represented by $A_j x \leq a_j$. We now define Φ_A as follows:

$$\Phi_A \equiv \forall j \in [1, N_A] \cdot \ \lambda_{A_j} \geq 0 \ \wedge \ \lambda_{A_j} A_j = i \ \wedge \ \lambda_{A_j} a_j \leq k$$

The row vector i is of the form $(i_{x_1} \ \cdots \ i_{x_n})$, where each i_{x_j} is a variable denoting the coefficient of variable x_j in the interpolant. Similarly, k is a variable denoting

the constant in the interpolant. Suppose we have an assignment to i, k, and λ_{A_j} that satisfies Φ_A. Then, by Farkas' lemma, the half-space $ix \leq k$ satisfies condition (1), where i and k are replaced by their assignment values. This because a satisfying assignment of Φ_A implies that for every $j \in [1, N_A]$, $A_j x \leq a_j \Rightarrow ix \leq k$.

We now encode a constraint Φ_B that enforces condition (2). As before, we associate a row vector λ_{B_j} with each sample $B_j x \leq b_j$. We define Φ_B as follows:

$$\Phi_B \equiv \forall j \in [1, N_B] \cdot \ \lambda_{B_j} \geq 0 \ \wedge \ \lambda_{B_j} B_j = -i \ \wedge \ \lambda_{B_j} b_j < -k$$

Following Farkas' lemma, a satisfying assignment of Φ_B results in a half-space $-ix < -k$ that subsumes all samples in s_B. That is, $B_j x \leq b_j \Rightarrow \neg ix \leq k$, for all $j \in [1, N_B]$, thus satisfying condition (2). Therefore, a satisfying assignment of $\Phi_A \wedge \Phi_B$ produces a half-space interpolant $ix \leq k$ for $(\bigvee s_A, \bigvee s_B)$. Note that our encoding implicitly ensures that coefficients of unshared variables are 0 in $ix \leq k$. See [6] for an example of solving these constraints to obtain an interpolant. Thm. 4 below states soundness and completeness of our encoding for samples that are conjunctions of closed half-spaces.

Theorem 4 (Soundness and Completeness of HalfItp). *Given two samplesets (where all samples are systems of closed half-spaces), s_A and s_B, and their corresponding encoding $\Phi \equiv \Phi_A \wedge \Phi_B$, then:*

1. *If Φ is satisfiable using a variable assignment m, then $i^m x \leq k^m$ is an interpolant for $(\bigvee s_A, \bigvee s_B)$, where i^m and k^m are the values of i and k in m, respectively.*
2. *Otherwise, no half-space interpolant exists for $(\bigvee s_A, \bigvee s_B)$.*

5 Implementation and Evaluation

We implemented the compositional SMT technique, Csmt, in the C♯ language, using the Z3 SMT solver [27] for constraint solving and sampling. The primary heuristic choice in the implementation is how to split and merge samplesets. The heuristics we use for this are described in [6], along with some optimizations that improve performance.

To experiment with Csmt, we integrated it with Duality [25], a tool that uses interpolants to construct inductive invariants. We will call this CsmtDua. In the configuration we used, Duality can be thought of as implementing *Lazy Abstraction With Interpolants* (LAWI), as in Impact [22]. The primary difference is that we use a large-block encoding [9], so that edges in the abstract reachability tree correspond to complete loop bodies rather than basic blocks.

Sequence Interpolants with Csmt. Duality produces sequences of formulas corresponding to (large block) program execution paths in static single assignment (SSA) form, and requires interpolants to be computed for these sequences. An interpolant sequence for formulas $A_1, \ldots, A_n$ is a sequence of formulas $I_1, \ldots, I_{n-1}$ where I_i is an interpolant for $(\bigwedge_{k \leq i} A_k, \bigwedge_{k > i} A_k)$. The interpolant sequence must also be be *inductive*, in the sense that $I_{i-1} \wedge A_i \Rightarrow I_i$. In

Program	CSMTDUA	CPA	UFO	INVGENAI	INVGENCS
f2	✔	✗	✗	$✗_f$	$✗_f$
gulv	✔	✗	✗	$✗_f$	$✗_f$
gulv_simp	✔	✔	✔	✔	✔
substring1	✔	✗	✔	✔	✔
pldi08	✔	✔	✔	$✗_f$	$✗_f$
pldi082unb	✔	✗	✗	✔	✔
xy0	✔	✗	✗	✔	✔
xy10	✔	✔	✔	✔	✔
xy4	✔	✗	✗	✔	✔
xyz	✔	✗	✗	✔	✔
xyz2	✔	✗	✗	✔	✔
dillig/01	✔	✔	✔	✔	✔
dillig/03	✔	✗	✗	✔	✔
dillig/05	✔	✗	✔	✔	✔
dillig/07	✔	✔	✔	✔	✔
dillig/09	✔	✗	✗	✔	✗
dillig/12	✔	✗	✔	✗	✗
dillig/15	✔	✔	✔	✔	✔
dillig/17	✔	✔	✗	✔	✔
dillig/19	✔	✔	✔	$✗_f$	$✗_f$
dillig/20	✔	✔	✔	$✗_f$	$✗_f$
dillig/24	✔	✔	✔	✔	✔
dillig/25	✔	✔	✔	✔	$✗_f$
dillig/28	✔	✗	✗	✔	✔
dillig/31	✔	✔	✗	✔	✗
dillig/32	✔	✔	✔	✔	✔
dillig/33	✔	✔	$✗_f$	✗	✗
dillig/35	✔	✔	✔	$✗_f$	$✗_f$
dillig/37	✔	✔	✔	$✗_f$	$✗_f$
dillig/39	✔	✔	✔	✔	✔
#SOLVED	**30**	17	17	21	18

Fig. 4. Verification results of CSMT, CPACHECKER, UFO, and two configurations of INVGEN on a collection of C benchmarks

this application, an inductive interpolant sequence can be thought of as a Hoare logic proof of the given execution path.

A key heuristic for invariant generation is to try to find a *common* interpolant for positions that correspond to the same program location. The requirement to find a simple common interpolant forces us to "fit" the emerging pattern of consecutive loop iterations, much as occurs in Figure 2(b). We can easily reduce the problem of finding a common interpolant to finding a single interpolant for a pair (A, B), provided we know the correspondence between variables in successive positions in the sequence.

Say we have substitutions σ_i mapping each program variables to its instance at position i in the SSA sequence. We construct formula A as $\bigvee_{i<n}(\bigwedge_{k\leq i} A_k)(\sigma_i^{-1})$ and B as $\bigvee_{i<n}(\bigwedge_{k>i} A_k)(\sigma_i^{-1})$. That is, A represents all the prefixes of the execution path and B all the suffixes. If I is an interpolant for (A, B), then the sequence $\{I_i\}$ where $I_i = I\sigma_i$ is an interpolant sequence for $A_1, \ldots, A_n$. If it is *inductive*, we are done, otherwise we abandon the goal of a common interpolant.

	iZ3		CSMT	
#ITERS	TIME(S)	SIZE	TIME(S)	SIZE
2	0.083	13	0.556	2
4	0.076	15	1.284	2
6	0.087	35	2.111	2
8	0.133	59	3.439	2
10	0.195	124	5.480	2
12	0.473	447	7.304	2
14	0.882	762	19.750	11
16	0.710	158	15.040	4
18	0.753	147	18.730	4
20	0.847	57	44.136	24
22	0.867	45	54.710	17
24	0.857	47	138.197	45
26	0.895	23	56.643	6
28	0.888	11	31.417	2
30	0.882	1	31.318	1

Fig. 5. Size of interpolants from CSMT and iZ3 for BMC unrollings of increasing length

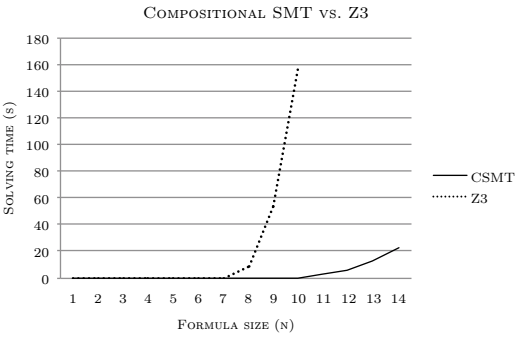


Fig. 6. Solving time (s) of CSMT and Z3 vs. formula size n

5.1 Evaluation

We now present an evaluation of CSMT. First, we compare CSMTDUA against a variety of verification tools on a set of benchmarks requiring non-trivial numerical invariants. Second, we demonstrate how our prototype implementation of CSMT can outperform Z3 at SMT solving. Finally, we compare the size of the interpolants computed by CSMT to those computed by iZ3 [23], an interpolating version of Z3.

CSMT for Verification. We compared CSMTDUA against the state-of-the-art verifiers CPACHECKER (CPA) [10] and UFO [5] – the top abstraction-based verifiers from the 2013 software verification competition (SV-COMP) [1]. CPA is a lazy abstraction tool that admits multiple configurations – we used its predicate abstraction and interpolation-based refinement configuration `predicateAnalysis`. UFO is an interpolation-based verification tool that guides interpolants with abstract domains [4,3] – we did not use any abstract domains for guidance (using an abstract domain does not change the number of solved benchmarks). We also compared against INVGEN [17], an invariant generation tool that combines non-linear constraint solving, numerical abstract domains, and dynamic analysis to compute safe invariants. We use INVGENAI to denote the default configuration of INVGEN, and INVGENCS to denote a configuration where abstract interpretation invariants are not used to strengthen invariants computed by constraint solving (`-nostrength` option).

To study the effectiveness of CSMTDUA, we chose small C programs from the verification literature that require non-trivial numeric invariants. The benchmarks `f2`, `gulv*`, and `substring1` are from [15]. The benchmarks `pldi08*` are from [16]. The benchmarks `xy*` are variations on a classic two-loop example requiring linear congruences. The benchmarks `dillig/*` were provided Dillig, *et al.* [13]. Some are from the literature and some are original and are

intended to test arithmetic invariant generation. We omitted benchmarks using the mod operator, as we do not support this, and also elided some duplicates and near-duplicates. The benchmarks and tool output are available at `http://www.cs.utoronto.ca/~aws/cav13.zip`.

Fig. 4 shows the result of applying CSMT and the aforementioned tools on 30 such benchmarks. In the table, ✔ means the tool verified the program, ✗ means that the tool timed out (after 60s), and $✗_f$ means the tool terminated unsuccessfully. The run times in successful cases tend to be trivial (less than 2s), so we do not report them. We observe that CSMTDUA produced proofs for all benchmarks, whereas CPA and UFO failed to prove 13, and INVGENAI failed to prove 9.

This shows the effectiveness of the heuristic that simple proofs tend to generalize. For example, in the case of `pldi082unb`, CSMTDUA produces the intricate invariant $x+y-2N \leq 2 \wedge y \leq x$. On the other hand, interpolant-based refinement in CPA and UFO diverges, producing an unbounded sequence of predicates only containing x and N.

Compositional SMT Solving. One way prove unsatisfiability of a formula $\Phi \equiv A \wedge B$ is to exhibit an interpolant I of (A, B). Using CSMT, this might be more efficient than direct SMT solving because (1) CSMT only makes SMT queries on the components A and B, and (2) by merging samples, CSMT can find proofs not available to the SMT solver's theory solver. This is especially true if Φ has many disjuncts.

Suppose, for example, we have

$$A \equiv x_0 = y_0 = 0 \wedge \bigwedge_{i=1}^{n} (\mathrm{inc}_i \vee \mathrm{eq}_i)$$

$$B \equiv \bigwedge_{i=n+1}^{2n} (\mathrm{dec}_i \vee \mathrm{eq}_i) \wedge x_{2n} = 0 \wedge y_{2n} \neq 0,$$

where inc_i is $x_i = x_{i-1} + 1 \wedge y_i = y_{i-1} + 1$, dec_i is $x_i = x_{i-1} - 1 \wedge y_i = y_{i-1} - 1$ and eq_i is $x_i = x_{i-1} \wedge y_i = y_{i-1}$. The formula $\Phi_n = A \wedge B$ is essentially the BMC formula for our benchmark `xy0`, where the two loops are unrolled n times. The pair (A, B) has a very simple interpolant, that is, $x_n = y_n$, in spite of the fact that each of A and B have 2^n disjuncts. Moreover, the conjunction $A \wedge B$ has 2^{2n} disjuncts. In a lazy SMT approach, each of these yields a separate theory lemma. Fig. 6 shows the time (in seconds) taken by Z3 and CSMT to prove unsatisfiability of Φ_n for $n \in [1, 14]$. For $n = 10$, Z3 takes 160 seconds to prove unsatisfiability of Φ_n, whereas CSMT requires 1.4 seconds. For $n > 10$, Z3 terminates without producing an answer. This shows that a compositional approach can be substantially more efficient in cases where a large number of cases can be summarized by a simple interpolant.

Interpolant Size. We now examine the relative complexity of the interpolants produced by CSMT and SMT-based interpolation methods, represented by iZ3 [23]. For our formulas, we used BMC unrollings of `s3_clnt_1`, an SSH benchmark from the software verification competition (SV-COMP) [1]. Fig. 5 shows the sizes of the

interpolants computed by iZ3 and CSMT (and the time taken to compute them) for unrollings (#ITER) of N iterations of the loop, with the interpolant taken after $N/2$ iterations. The size of the interpolant is measured as the number of operators and variables appearing in it. Since the loop has a reachability depth of 14, all the interpolants from $N = 30$ are *false*. For $N = 12$, the interpolant size for iZ3 is 447, whereas for CSMT it is 2, a 200X reduction. A large reduction in interpolant size is observed for most unrolling lengths. Notice that, since this example has few execution paths, SMT is quite fast. This illustrates the opposite case from the previous example, in which the compositional approach is at a significant performance disadvantage.

6 Related Work

We compare CSMT against interpolation and invariant generation techniques.

Constraint-Based Techniques. In [29], Rybalchenko and Stokkermans describe a method of for computing half-spaces separating two convex polytopes using Farkas' lemma. Here, we generalize this method to separators for sets of polytopes. This helps us search for a simple interpolant separating all the samples, rather than constructing one separating plane for each sample pair. This in turn gives us an interpolation procedure for arbitrary formulas in QFLRA, rather than just conjunctions of literals.

Interpolants from Classifiers. Our work is similar in flavor to, and inspired by, an interpolant generation approach of Sharma et al. [31]. This approach uses point samples of A and B (numerical satisfying assignments) rather than propositional disjuncts. A machine learning technique – *Support Vector Machines* (SVM's) – is used to create linear separators between these sets. The motivation for using disjuncts (polytopes) rather than points is that they give a broader view of the space and allow us to exploit the logical structure of the problem. In particular, it avoids the difficult problem of clustering random point samples in a meaningful way. In practice, we found that bad clusterings led to complex interpolants that did not generalize. Moreover, we found the SVM's to be highly sensitive to the sample set, in practice often causing the interpolation procedure to diverge, *e.g.*, by finding samples that approach the boundary of a polytope asymptotically.

Interpolants from Refutation Proofs. A number of papers have studied extracting "better" interpolants from refutation proofs. For example, [14,32,28] focused on the process of extracting interpolants of varying strengths from refutation proofs. Hoder et al. [19] proposed an algorithm that produces syntactically smaller interpolants by applying transformations to refutation proofs. Jhala and McMillan [20] described a modified theory solver that yields interpolants in a bounded language. In contrast, we have taken the approach of structuring the proof search process expressly to yield simple interpolants. Our method can compute simpler and more general interpolants, by discovering facts that are not found in refutation proofs produced by lazy SMT solvers. On the other hand, constructing interpolants from refutation proofs can be much faster in

cases where the number of theory lemmas required is small. Also, while the compositional approach may be applicable to the various theories handled by proof-based interpolation, we have as yet only developed a method for LRA.

Template Methods. A more direct approach to synthesize linear inductive invariants is based on Farkas' Lemma and non-linear constraint solving [12]. The invariant is expressed as a fixed conjunction of linear constraints with unknown coefficients, and one solves simultaneously for the invariant and the Farkas proof of its correctness. This has the advantage, relative to the interpolant approach, that it does not require unfolding the program and the disadvantage that it requires a non-linear arithmetic solver. Currently, such solvers do not scale to more than a few variables. Thus, the difficulty of finding a solution grows rapidly with the number of constraints in the invariant [12]. An example of a tool using this approach is InvGen, run without abstract interpretation (called InvGenCS in Table 4). An examination of the 12 cases in which InvGenCS fails shows that in most the invariant we found has at least three conjuncts, and in some it is disjunctive, a case that the authors of InvGen have found impractical using the method [17]. Thus, it appears that by searching for simple interpolants, we can synthesize invariants with greater propositional complexity than can be obtained using the constraint-based approach.

7 Conclusion

We have developed a compositional approach to interpolation based on the heuristic that simpler proofs of special cases are more likely to generalize. The method produces simple (perhaps even beautiful) interpolants because it is able to summarize a large set of cases using one relatively simple fact. In particular, we presented a method for finding such simple facts in the theory of linear rational arithmetic. This made it possible to use interpolation to discover inductive invariants for numerical programs that are challenging for existing techniques. We also observed that for formulas with many disjuncts, the compositional approach can be more efficient than non-compositional SMT.

Our work leaves many avenues open for future research. For example, can the method be effectively applied to integer arithmetic, or the theory of arrays? From a scalability standpoint, we would like to improve the performance of Csmt on formulas requiring more complex interpolants. One possible direction is parallelism, e.g., instead of decomposing a formula into $A \wedge B$, we could decompose it into multiple sets of conjuncts and use techniques such as [18] to parallelize SMT solving.

References

1. Competition On Software Verification, `http://sv-comp.sosy-lab.org/`
2. Merriam-Webster Dictionary (December 2012), `www.merriam-webster.com`
3. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)

4. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From under-approximations to over-approximations and back. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 157–172. Springer, Heidelberg (2012)
5. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
6. Albarghouthi, A., McMillan, K.L.: Beautiful interpolants. Tech. Rep. MSR-TR-2013-42, Microsoft Research (April 2013)
7. Alur, R., Dang, T., Ivančić, F.: Counter-example guided predicate abstraction of hybrid systems. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 208–223. Springer, Heidelberg (2003)
8. Barrett, C.W., Sebastiani, R., Seshia, S.A., Tinelli, C.: Satisfiability modulo theories. In: Handbook of Satisfiability (2009)
9. Beyer, D., Cimatti, A., Griggio, A., Keremoglu, M.E., Sebastiani, R.: Software Model Checking via Large-Block Encoding. In: FMCAD 2009, pp. 25–32 (2009)
10. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
11. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient Generation of Craig Interpolants in Satisfiability Modulo Theories. ACM Trans. Comput. 12(1), 7 (2010)
12. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
13. Dillig, I., Dillig, T., Li, B.: Personal Communication (2012)
14. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
15. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
16. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI 2008, pp. 281–292 (2008)
17. Gupta, A., Rybalchenko, A.: InvGen: An efficient invariant generator. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 634–640. Springer, Heidelberg (2009)
18. Hamadi, Y., Marques-Silva, J., Wintersteiger, C.M.: Lazy decomposition for distributed decision procedures. In: PDMC 2011, pp. 43–54 (2011)
19. Hoder, K., Kovács, L., Voronkov, A.: Playing in the grey area of proofs. In: POPL 2012, pp. 259–272 (2012)
20. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
21. Kroening, D., Weissenbacher, G.: Lifting propositional interpolants to the word-level. In: FMCAD 2007, pp. 85–89 (2007)
22. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
23. McMillan, K.L.: Interpolants from Z3 proofs. In: FMCAD 2011, pp. 19–27 (2011)
24. McMillan, K.L.: An interpolating theorem prover. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 16–30. Springer, Heidelberg (2004)

25. McMillan, K.L., Rybalchenko, A.: Computing relational fixed points using interpolation. Tech. Rep. MSR-TR-2013-6, Microsoft Research (2013)
26. Megiddo, N.: On the complexity of polyhedral separability. Discrete & Computational Geometry 3, 325–337 (1988)
27. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
28. Rollini, S.F., Sery, O., Sharygina, N.: Leveraging interpolant strength in model checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 193–209. Springer, Heidelberg (2012)
29. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. 45(11), 1212–1233 (2010)
30. Schrijver, A.: Theory of linear and integer programming. John Wiley & Sons, Inc., New York (1986)
31. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
32. Weissenbacher, G.: Interpolant strength revisited. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 312–326. Springer, Heidelberg (2012)

Efficient Generation of Small Interpolants in CNF

Yakir Vizel[1], Vadim Ryvchin[2,3], and Alexander Nadel[3]

[1] Computer Science Department, The Technion, Haifa, Israel
[2] Information Systems Engineering Department, The Technion, Haifa, Israel
[3] Design Technology Solutions Group, Intel Corporation, Haifa, Israel

Abstract. Interpolation-based model checking (ITP) [14] is an efficient and complete model checking procedure. However, for large problems, interpolants generated by ITP might become extremely large, rendering the procedure slow or even intractable.

In this work we present a novel technique for interpolant generation in the context of model checking. The main novelty of our work is that we generate *small* interpolants in *Conjunctive Normal Form (CNF)* using a twofold procedure: First we propose an algorithm that exploits resolution refutation properties to compute an interpolant approximation. Then we introduce an algorithm that takes advantage of inductive reasoning to turn the interpolant approximation into an interpolant. Unlike ITP, our approach maintains only the relevant subset of the resolution refutation. In addition, the second part of the procedure exploits the properties of the model checking problem at hand, in contrast to the general-purpose algorithm used in ITP.

We developed a new interpolation-based model checking algorithm, called CNF-ITP. Our algorithm takes advantage of the smaller interpolants and exploits the fact that the interpolants are given in CNF. We integrated our method into a SAT-based model checker and experimented with a representative subset of the HWMCC'12 benchmark set. Our experiments show that, overall, the interpolants generated by our method are 42 times smaller than those generated by ITP. Our CNF-ITP algorithm outperforms ITP, and at times solves problems that ITP cannot solve. We also compared CNF-ITP to the successful IC3 [3] algorithm. We found that CNF-ITP outperforms IC3 [3] in a large number of cases.

1 Introduction

Model checking is a method for formally verifying that a system satisfies a predefined set of properties. A SAT-solver is a powerful decision procedure used in model checking. While in the early days SAT-based model checking was only used for bug-hunting, nowadays it is a complete procedure and can either prove or refute properties. One such complete SAT-based algorithm uses *Interpolation* [14].

We present a novel approach for interpolant computation in the context of SAT-based model checking. The main contribution of this work is the ability to produce *small* interpolants in *Conjunctive Normal Form* (CNF) efficiently. In order to compute an interpolant, our work takes advantage both of the properties of the resolution refutation, generated by the SAT solver, and of the structure of the model checking problem at hand. In addition, we present *CNF-ITP*, an enhanced version of the original

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 330–346, 2013.

interpolation-based model checking algorithm [14] (ITP). CNF-ITP makes use of the fact that interpolants are given in CNF.

Given a pair of inconsistent propositional formulas $A(X,Y)$ and $B(Y,Z)$, where X, Y and Z are sets of Boolean variables, an interpolant $I(Y)$ is a formula that fulfills the following properties: $A(X,Y) \Rightarrow I(Y)$; $I(Y) \wedge B(Y,Z)$ is unsatisfiable; and $I(Y)$ is a formula over the common variables of $A(X,Y)$ and $B(Y,Z)$ [6]. Modern SAT-solvers are capable of generating an unsatisfiability proof of an unsatisfiable formula. The proof is in the form of a resolution refutation [22,10,16]. It is possible to compute an *interpolant* from a resolution refutation of $A(X,Y) \wedge B(Y,Z)$ [17,14].

Interpolants are used in various domains. The work in [14] was the first to incorporate interpolants into model checking, creating a complete SAT-based algorithm referred to as ITP. ITP uses interpolants to over-approximate image computations. Since [14], interpolants have been applied in several model checking algorithms [11,12,15,20,21].

[14] presents a recursive procedure for interpolant generation from a proof. The procedure initially assigns a propositional formula to each one of the leaves in the resolution refutation (hypothesis clauses). It then recursively assigns a propositional formula to every node in the refutation by either conjoining or disjoining the propositional formulas of its predecessors. Choosing between conjunction or disjunction depends on whether the pivot variable is local to $A(X,Y)$ or not. The formula that is assigned for the empty clause represents the interpolant.

While this algorithm is linear in the size of the proof, the resulting interpolant is a non-CNF propositional formula that mirrors the structure of the resolution refutation. Thus, when the resolution refutation is large, so is the interpolant. Moreover, the resulting formula is often highly redundant, meaning that the interpolant can be simplified and be represented by a smaller formula.

ITP requires the interpolants to be fed back into the SAT solver for computing the next interpolant. Therefore, in those cases where the size of interpolants is large, the resulting SAT problem may be intractable.

We strive to solve this problem by natively generating small interpolants in CNF. One way to compute an interpolant is by *existential quantification*. Considering the unsatisfiable formula $A(X,Y) \wedge B(Y,Z)$, $I(Y) = \exists X(A(X,Y))$ is an interpolant. For a CNF formula $A(X,Y)$, $\exists X(A(X,Y))$ can be created by iteratively applying variable elimination[1] on X variables in $A(X,Y)$. The problem with this approach is that variable elimination is exponential, and, therefore impractical, given a large set of variables.

In this work, we provide a novel resolution-refutation-guided method for variable elimination to derive an interpolant in CNF. This procedure, while creating less clauses than naïve variable elimination procedures, might still result in an exponential blow-up.

Our solution is first to build an *approximated* interpolant $I_w(Y)$ for which $I_w(Y) \wedge B(Y,Z)$ may be satisfiable. We refer to such an interpolant as a B_{weak}*-interpolant*. Computing the B_{weak}-interpolant is based on the method of resolution-refutation-guided variable elimination but is far more efficient. The second stage of our method aims

[1] *Variable elimination* [7] is an operation that replaces all occurrences of a variable v from a CNF formula by replacing clauses containing v with the result of pairwise resolutions between clauses containing the literal v and those containing the literal $\neg v$.

at strengthening $I_w(Y)$ and transforming it into an interpolant $I(Y)$ where $I(Y) \wedge B(Y,Z)$ is unsatisfiable. We refer to this process as *B-Strengthing*.

In order to transform a B_{weak}-interpolant into an interpolant we need to make sure that $A(X,Y) \Rightarrow I_w(Y)$ and that $I_w(Y) \wedge B(Y,Z)$ is unsatisfiable. This can be done by finding all satisfying assignments $s(Y)$ to $I_w(Y) \wedge B(Y,Z)$ and conjoining $\neg s(Y)$ with $I_w(Y)$. Note that an assignment s is a conjunction of literals, and therefore its negation is a clause. By this we keep $I_w(Y)$ in CNF. The number of such assignments may be vast, and therefore this is an inefficient method.

To overcome this, instead of adding a clause to $I_w(Y)$ we *generalize* it to a sub-clause so as to block a larger set of assignments. In order to perform an efficient generalization we use the structure of A. In the context of model checking, $A(V,V') = Q(V) \wedge TR(V,V')$ where V is the set of variables in the checked system and *TR* is the transition relation. Using this fact allows us to perform *inductive generalization* [3].

We implemented CNF-ITP, a model checking algorithm which is a variant of ITP [14], but which uses the above method to compute the interpolants. Our goal was to measure the impact of our interpolant computation method on the underlying model checking algorithm. However, CNF-ITP also exploits the fact that interpolants are given in CNF in order to improve the traditional ITP. Our improvements to ITP were inspired by [3].

For the experiments we used the HWMCC'12 benchmark set. The interpolants computed by our method, compared to those computed by the original ITP algorithm of [14], were much smaller in size overall in the vast majority of cases. Sometimes, the size was up to *two* orders of magnitude smaller. Our procedure significantly outperformed ITP and solved some test cases that ITP could not solve. To complete our experiments, we also compared CNF-ITP to the successful IC3 [3] algorithm. We found that CNF-ITP outperformed IC3 [3] in a large number of cases.

1.1 Related Work

A well-known problem of interpolants is their size. Several works try to deal with this problem. The work in [4] suggests dealing with the increasing size of interpolants by using circuit compaction. While this process can be efficient in some cases, it may consume considerable resources for very large interpolants. Moreover, compacting an interpolant does not result in a CNF formula, whereas our approach results in interpolants in CNF.

As we have already noted, an interpolant computed from a resolution refutation mirrors its structure. Several works [1,18] deal with reductions to the resolution refutation. Since our method uses resolution refutation it too can benefit from such an approach.

During interpolant computation, our approach only uses the relevant parts of the resolution refutation. The idea of holding and maintaining only the relevant parts of the resolution derivation was proposed and proved useful in [19] in the context of group-oriented minimal unsatisfiable core extraction.

Deriving interpolants in CNF was suggested in [12]. The authors suggest applying a set of reordering rules for resolution refutations so that the resulting interpolant will be in CNF. As the authors state in the paper, the described procedure does not always return an interpolant in CNF. Also, the reordering of a resolution refutation may result in an exponential blow up of the proof and, as stated in [8], reordering is not always possible.

In contrast to [12], our method does not rewrite the resolution refutation generated by the SAT solver.

The work in [5] suggests an interpolant computation method that does not use the generated resolution refutation. In addition, an interpolant that results from the use of that method is in a Disjunctive Normal Form (DNF). Our work, on the other hand, uses the resolution refutation and generates interpolants in CNF efficiently.

2 Preliminaries

Throughout the paper we denote the value *false* as $\bot$ and the value *true* as $\top$.

Let V be a set of Boolean variables. For $v \in V$, v' is used to denote the value of v after one time unit. The set of these variables is denoted by V'. In the general case V^i is used to denote the variables in V after i time units (thus, $V^0 = V$). For a propositional formula F over V we write F' to denote the same formula when substituting every occurrence of $v \in V$ in F with $v' \in V'$. In the general case, we write $F(V^i)$ to denote the substitution of every occurrence of $v^j \in V^j$ in F with $v^i \in V^i$ for some non-negative i, j. From now on, all formulas we refer to are *propositional formulas*, unless stated otherwise.

Definition 1. *A* finite transition system *is a triple* $M = (V, \mathrm{INIT}, \mathrm{TR})$ *where* V *is a set of boolean variables,* $\mathrm{INIT}(V)$ *is a formula over* V*, describing the initial states, and* $\mathrm{TR}(V, V')$ *is a formula over* V *and the next-state variables* V'*, describing the transition relation.*

In order to describe a path in a transition system M by means of propositional formulae we define: $path^{i,j} = TR(V^i, V^{i+1}) \wedge \ldots \wedge TR(V^{j-1}, V^j)$ where $0 \leq i < j$. Abusing notation somewhat, we sometimes refer to a propositional formula over V as a set of states in M.

Definition 2 (Conjunctive Normal Form (CNF)). *Given a set* U *of Boolean variables, a literal* l *is a variable* $u \in U$ *or its negation and a* clause *is a set of literals. A formula* F *in CNF is a conjunction of clauses.*

A SAT solver is a complete decision procedure that, given a set of clauses, determines whether the clause set is *satisfiable* or *unsatisfiable*. A clause set is said to be satisfiable if there exists a *satisfying assignment* such that every clause in the set is evaluated to $\top$. If the clause set is satisfiable then the SAT solver returns a satisfying assignment for it. Otherwise the solver produces a *resolution refutation* comprising the proof of unsatisfiability [22,10,16].

For a formula X, $\mathcal{V}(X)$ is the set of variables appearing in X.

Definition 3 (Local and Global Variable). *Let* (A, B) *be a pair of formulas in CNF. A variable* v *is* A*-local (*B*-local) iff* $v \in \mathcal{V}(A) \backslash \mathcal{V}(B)$ *(*$v \in \mathcal{V}(B) \backslash \mathcal{V}(A)$*);* v *is* (A, B)*-global or, simply, global, iff* $v \in \mathcal{V}(A) \cap \mathcal{V}(B)$*.*

Definition 4 (Interpolant). *Let* (A, B) *be a pair of formulas in CNF such that* $A \wedge B \equiv \bot$*. The* interpolant *for* (A, B) *is a formula* I *such that: (i)* $A \Rightarrow I$*. (ii)* $I \wedge B \equiv \bot$*. (iii)* $\mathcal{V}(I) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$ *(all the variables in the interpolant are global).*

We will use the notions of weaker versions of interpolants that fulfill two out of three interpolant properties.

Definition 5 (B_{weak}-Interpolant). *Let (A, B) be a pair of formulas in CNF such that $A \wedge B \equiv \bot$. The* B_{weak}-interpolant *for (A, B) is a formula I such that: (i) $A \Rightarrow I$. (ii) $\mathcal{V}(I) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$.*

Definition 6 (Non-Global-Interpolant). *Let (A, B) be a pair of formulas in CNF such that $A \wedge B \equiv \bot$. The* non-global-interpolant *for (A, B) is a formula I such that: (i) $A \Rightarrow I$. (ii) $I \wedge B \equiv \bot$.*

Next we provide some resolution-related definitions. The *resolution rule* states that given clauses $\alpha_1 = \beta_1 \vee v$ and $\alpha_2 = \beta_2 \vee \neg v$, where β_1 and β_2 are also clauses, one can derive the clause $\alpha_3 = \beta_1 \vee \beta_2$. Application of the resolution rule is denoted by $\alpha_3 = \alpha_1 \otimes^v \alpha_2$.

Definition 7 (Resolution Derivation). *A* resolution derivation *of a target clause α from a CNF formula $G = \{\alpha_1, \alpha_2, \ldots, \alpha_q\}$ is a sequence*
$\pi = (\alpha_1, \alpha_2, \ldots, \alpha_q, \alpha_{q+1}, \alpha_{q+2}, \ldots, \alpha_p \equiv \alpha)$, where each clause α_i for $i \leq q$ is initial *and α_i for $i > q$ is* derived *by applying the resolution rule to α_j and α_k, where $j, k < i$.*

A resolution derivation π can naturally be conceived of as a directed acyclic graph (DAG) whose vertices correspond to all the clauses of π and in which there is an edge from a clause α_j to a clause α_i iff $\alpha_i = \alpha_j \otimes \alpha_k$. A clause $\beta \in \pi$ is a *parent* of $\alpha \in \pi$ iff there is an edge from β to α. A clause $\beta \in \pi$ is *backward reachable* from $\gamma \in \pi$ if there is a path (of 0 or more edges) from β to γ. The set of all vertices backward reachable from $\beta \in \pi$ is denoted $\Gamma(\pi, \beta)$.

Definition 8 (Resolution Refutation). *A resolution derivation π of the empty clause $\square$ from a CNF formula G is called the resolution refutation of G.*

An interpolant can be produced out of a resolution refutation [14].

For this work, we will need a definition of an A-resolution refutation, that is, a projection of a given resolution refutation π to the clause set A:

Definition 9 (A-Resolution Refutation). *Let $\pi = (\alpha_1, \alpha_2, \ldots, \square)$ be a resolution refutation of the CNF formula $G = A \wedge B$. The* A-resolution refutation *$\pi_A \in \pi$ is constructed by applying the following operation for every clause $\alpha_i \in \pi$ in the order of appearence in π: α_i is appended to π_A iff either $\alpha_i \in A$ or $\alpha_i = \alpha_j \otimes^v \alpha_k$ and $\alpha_j \in \pi_A$ or $\alpha_k \in \pi_A$.*

In DAG terminology π_A is a sub-graph of π that contains only those vertices whose clauses belong to A, and the edges between such clauses. Note that a clause $\alpha \in \pi$ may have 0 or 2 parents, while a clause $\alpha \in \pi_A$ may also have 1 parent (if the second parent is implied only by the clauses of B).

We denote clauses containing the literal v/$\neg v$ in a given clause set by v^+/v^-, respectively. Given a CNF formula F and a variable $v \in \mathcal{V}(F)$, *variable elimination* [7] is an

operation that removes v from F by replacing clauses containing the variable v with the result of a pairwise resolution between v^+ and v^-. The resulting formula $VE(F, v)$ is equisatisfiable with F [7]. The groundbreaking DP algorithm for deciding propositional satisfiability [7] uses variable elimination until either the empty clause $\square$ is derived, in which case the formula is unsatisfiable, or all the variables appear in one polarity only, in which case the formula is satisfiable. It is well known that the original DP algorithm suffers from exponential blow-up.

A bounded version of variable elimination has been an essential contributor to the efficiency of modern SAT preprocessing algorithms (that is, algorithms that truncate the size of the CNF formula before embarking on the search) since the introduction of the SatELite preprocesor [9]. In *bounded variable elimination*, used in SatELite, a variable v is eliminated iff the operation does not increase the number of clauses.

3 Generating Interpolant Approximation in CNF

In this section we propose a method for generating a B_{weak}-interpolant (recall Def. 5) in CNF. First, we briefly describe two algorithms for generating interpolants in CNF. In practice, both algorithms are not applicable to all cases, because of exponential blow-up. Thereafter we introduce an efficient algorithm which is guaranteed to return a B_{weak}-interpolant in CNF, and which may for some cases return an interpolant in CNF.

Our first algorithm for generating an interpolant in CNF is based on naïve variable elimination. First it generates a resolution refutation of the given formula using a SAT solver. Then it initializes the interpolant by those clauses of A that are backward reachable from $\square$ (the empty clause). Note that at this stage I is a non-global-interpolant (recall Def. 6). Finally, the algorithm gradually turns the non-global-interpolant into an interpolant by applying variable elimination over all A-local variables. Consider the example in Fig. 1. Our algorithm would generate the following interpolant: $I = \{g_1 \vee g_2, g_1 \vee g_4, g_3 \vee g_2, g_3 \vee g_4\}$. Unfortunately, the algorithm suffers from the same drawback as the DP algorithm [7]: exponential blow-up when variables keep being eliminated.

Our next algorithm is based on the observation that to eliminate a variable v it is not necessary to apply resolution over all the pairs in v^+ and v^-, but rather only over those subsets that contribute to deriving a common ancestor in the resolution derivation. We need to introduce the notion of clause-interpolant.

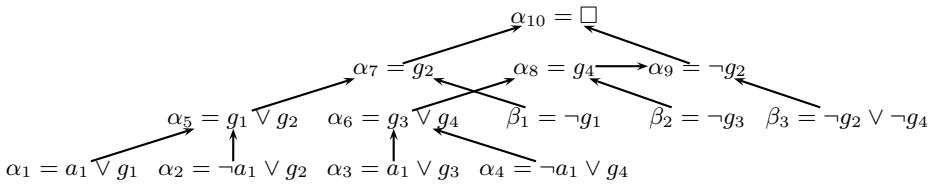


Fig. 1. An example of a resolution refutation. Assume $A = \{\alpha_1, \ldots, \alpha_4\}$ and $B = \{\beta_1, \ldots, \beta_3\}$.

Definition 10 (Clause-Interpolant). *Let (A, B) be an unsatisfiable pair of CNF formulas. Let α be a clause. Then, $I(\alpha)$ is a* Clause-Interpolant *of α iff:*
(i) $A \Rightarrow I(\alpha)$ (ii) $I(\alpha) \wedge B \Rightarrow \alpha$ (iii) $\mathcal{V}(I(\alpha)) \subseteq (\mathcal{V}(A) \cap \mathcal{V}(B)) \cup (\mathcal{V}(A) \cap \mathcal{V}(\alpha))$

A clause-interpolant is a generalization of an interpolant that allows one to associate an interpolant with every clause α in A-resolution refutation (recall Def. 9). As in the case of the standard interpolant, the clause-interpolant is implied by A. The conjunction of the clause-interpolant with B implies the clause α (instead of $\square$ for the standard interpolant). Finally, the clause-interpolant is allowed to contain global variables and A-local variables that appear in α. Note that a clause-interpolant of $\square$ is an interpolant.

Our second proposed algorithm for deriving an interpolant in CNF works as follows: it traverses the A-resolution refutation from the input clauses towards $\square$. It constructs a clause-interpolant for each traversed clause as follows. The clause-interpolant of each initial clause α is set to $\{\alpha\}$. For creating the clause-interpolant of a derived clause α, the algorithm first conjoins the clause-interpolants of α's parents. Then, if α was created by resolution over a local variable v, v is eliminated from the result. The clause-interpolant of $\square$ is returned as the interpolant. Consider again the example in Fig. 1. We have $I(\alpha_5) = \alpha_1 \otimes^{a_1} \alpha_2 = g_1 \vee g_2$; $I(\alpha_6) = \alpha_3 \otimes^{a_1} \alpha_2 = g_3 \vee g_4$; $I(\alpha_7) = I(\alpha_5)$; $I(\alpha_9) = I(\alpha_8) = I(\alpha_6)$. Finally, the interpolant is $I(\square) = I(\alpha_7) \cup I(\alpha_9) = \{g_1 \vee g_2, g_3 \vee g_4\}$. Note that for our example, the interpolant generated by the current algorithm is smaller than the one generated by our previous algorithm, which applies exhaustive variable elimination. In practice, however, the current algorithm is not always scalable either, due to the same problem – exponential blow-up caused by variable elimination. Also note that for our simple example the intepolant comprises a cut $\{\alpha_5, \alpha_6\}$ in the A-resolution refutation, where all the clauses are implied by A only. One can show that whenever such a cut exists it comprises an interpolant. Unfortunately, in the general case such cuts do not usually exist.

Now we are ready to present a scalable algorithm for approximating an interpolant by generating a B_{weak}-interpolant. The first stage of our algorithm traverses the resolution refutation to generate a non-global-interpolant. The second stage uses bounded variable elimination and then incomplete variable elimination (defined below), if required, to convert the non-global-interpolant to a B_{weak}-interpolant.

Definition 11 (Incomplete Variable Elimination). *Given a CNF formula F and a variable $v \in \mathcal{V}(F)$,* incomplete variable elimination *is an operation that removes v from F by replacing clauses containing the variable v with the set IVE(F, v) which contains* some *of the results of a pairwise resolution between v^+ and v^-, where two requirements are met:*

1. *$|IVE(F, v)| \leq |v^+| + |v^-|$*
2. *Let $\alpha \in v^+/v^-$ be a clause, such that there exists a clause $\beta \in v^-/v^+$, such that $\alpha \otimes^v \beta$ is not a tautology. Then, $\alpha \otimes^v \gamma \in IVE(F, v)$ for* at least one *clause $\gamma \in v^-/v^+$, such that $\alpha \otimes^v \gamma$ is not a tautology.*

The idea behind incomplete variable elimination is to omit some of the resolvents when eliminating the variable v in order not to increase the number of clauses, yet to guarantee that each clause containing v has some contribution to the generated set of clauses. Note

that while incomplete variable elimination is not sufficient to maintain unsatisfiability for all cases, it may be sufficient for some cases. Incomplete variable elimination is non-deterministic.

Before presenting our eventual algorithm, we need to introduce the notion of a non-global-clause-interpolant:

Definition 12 (Non-Global-Clause-Interpolant). *Let* (A, B) *be an unsatisfiable pair of CNF formulas. Let* α *be a clause. Then,* $I(\alpha)$ *is a* Non-Global-Clause-Interpolant *of* α *iff: (i)* $A \Rightarrow I(\alpha)$ *(ii)* $I(\alpha) \wedge B \Rightarrow \alpha$

Note that a non-global-clause-interpolant of $\square$ is a non-global-interpolant.

Consider now the algorithm described in Fig. 2. Its first part (lines 2-21) traverses the resolution refutation and asssociates a non-global-clause-interpolant with each clause. Consider a visited clause $\alpha_i = \alpha_j \otimes^v \alpha_k$ when v is local. First, the algorithm sets $I(\alpha_i)$ to be the union of $I(\alpha_j)$ and $I(\alpha_k)$. It eliminates the variable v if the following two conditions hold: First, that eliminating v does not increase the clause size of $I(\alpha_i)$ (as in the bounded variable elimination of SatELite [9]), and second, that variable elimination has been performed for all clauses backward reachable from α_i. (The second condition is ensured by using an auxiliary set *Skipped* for marking clauses for which variable elimination was skipped). The second stage of the algorithm (starting from line 22) uses bounded variable elimination and then incomplete variable elimination to convert the non-global-interpolant to the eventually returned B_{weak}-interpolant by eliminating A-local variables. Note that the bounded variable elimination stage is non-redundant even

```
1: function SIG(π_A = (α_1, α_2, ..., α_q, α_{q+1}, α_{q+2}, ..., α_p ≡ □))
2:     Skipped := {}
3:     for all i ∈ {1, 2, ..., q} do
4:         I(α_i) := {α_i}
5:     end for
6:     for all i ∈ {q + 1, q + 2, ..., p ≡ □} do
7:         if α_i has exactly one parent β then
8:             I(α_i) := I(β)
9:         else
10:            if α_i = α_j ⊗^v α_k, where v is global then
11:                I(α_i) := I(α_j) ∪ I(α_k)
12:            else      // α_i = α_j ⊗^v α_k, where v is A-local
13:                I(α_i) := I(α_j) ∪ I(α_k)
14:                if |VE(I(α_j) ∪ I(α_k), v)| ≤ |I(α_j) ∪ I(α_k)| and {α_j, α_k} ∩ Skipped = ∅ then
15:                    I(α_i) := VE(I(α_i), v)
16:                else
17:                    Skipped := Skipped ∪ {α_i}
18:                end if
19:            end if
20:        end if
21:    end for
22:    Apply bounded variable elimination for A-local variables over I(□)
23:    if I(□) then do not contain A-local variables
24:        return I(□)       // In this case I(□) is an interpolant
25:    else
26:        Apply incomplete variable elimination for A-local variables over I(□)
27:        return I(□)       // In this case I(□) is a B_weak-interpolant
28:    end if
29: end function
```

Fig. 2. B_{weak}-Interpolant Generation

```
1: function ITP(M,p)
2:     if INIT ∧ ¬p == SAT then
3:         return cex
4:     end if
5:     k = 1
6:     while true do
7:         result = ComputeReachable(M, p, k)
8:         if result == fixpoint then
9:             return Valid
10:        else if result == cex then
11:            return cex
12:        end if
13:        k = k + 1
14:    end while
15: end function
```

Fig. 3. Interpolation-Based Model Checking (ITP)

though bounded variable elimination was performed locally for resolution refutation clauses, since sometimes bounded variable elimination is possible given a large set of clauses while it is impossible given a subset of that set. Note also that the algorithm returns an interpolant rather than merely a B_{weak}-interpolant if all the A-local variables are succesfully removed before incomplete variable elimination is applied.

4 Using B_{weak}-Interpolants In Model Checking

In this section we describe a model checking algorithm that uses B_{weak}-interpolants. Our algorithm is composed of two main stages. Recall that by Def. 5, a B_{weak}-interpolant fulfills two out of the three conditions of an interpolant. Therefore, the first stage attempts to transform the B_{weak}-interpolant into an interpolant.

The second stage uses interpolants computed by the first stage. In essence, the second stage is a modification of the original ITP and is called CNF-ITP. Besides the fact that CNF-ITP uses interpolants in CNF, it further takes advantage of this fact by applying optimizations which are possible only as a result of using interpolants in CNF.

Before going into the details of CNF-ITP, we describe ITP.

4.1 Interpolation-Based Model Checking Revisited

ITP [14] is a complete SAT-based model checking algorithm. It uses interpolation to over-approximate the reachable states in a transition system M with respect to a property p. ITP uses nested loops where the outer loop increases the depth of unrolling and the inner loop computes the reachable states. ITP is described in Fig. 3

Definition 13. *Let k and n be the depth of unrolling used in the outer loop and the iteration of the inner loop of ITP respectively. We define $R_n^k = \text{INIT} \vee I_1^k \vee I_2^k \vee \ldots \vee I_n^k$ to be the set of reachable states computed by the inner loop of ITP after n iterations and with respect to unrolling depth k. For a given $1 \leq j \leq n$, I_j^k is the interpolant computed in the j-th iteration of the inner loop.*

From this point and on, k and n refer to the depth of unrolling used in the outer loop and the iteration of the inner loop of ITP respectively.

16: **function** COMPUTEREACHABLE(M,p, k)
17: $R_0^k = INIT$, $I_0^k = INIT$, $n = 1$
18: **if** $I_0^k \wedge \text{path}^{0,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$ == SAT **then**
19: **return** cex
20: **end if**
21: **repeat**
22: $A = I_{n-1}^k(V^0) \wedge TR(V^0, V^1)$
23: $B = \text{path}^{1,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$
24: $I_n^k = \text{GETINTERPOLANT}(A, B)$
25: **if** $I_n^k \Rightarrow R_{n-1}^k$ **then**
26: **return** $fixpoint$
27: **end if**
28: $R_n^k = R_{n-1}^k \vee I_n^k$
29: $n = n + 1$
30: **until** $I_{n-1}^k \wedge \text{path}^{0,k} \wedge (\neg p(V^1) \vee \ldots \vee \neg p(V^k))$ == SAT
31: **end function**

Fig. 4. Inner loop of ITP

In general, the inner loop checks a fixed-bound BMC [2] formula where at each iteration only the initial states are replaced with an interpolant computed at a previous iteration (line: 30). This is done until the BMC formula becomes SAT (line: 30) or until a fixpoint is reached (lines: 25-27). In the former case, the outer loop increases the unrolling depth by 1[2] (line: 13) in order to either increase the precision of the over-approximations or to find a counterexample.

Lemma 1. $R_n^k(V^0) \wedge path^{0,k-1} \wedge (\bigvee_{j=0}^{k-1} \neg p(V^j))$ *is unsatisfiable.*

The above lemma is derived directly from the interpolant definition and from the way R_n^k is computed in ITP. R_n^k is also referred to as $(k-1)$-adequate.

4.2 Transforming a B_{weak}-Interpolant into an Interpolant Using Inductive Reasoning

As we have shown in Sec. 3, given a pair of formulas (A, B) such that $A \wedge B$ is unsatisfiable, a B_{weak}-interpolant I_w can be computed. By Def. 5, $A \Rightarrow I_w$ and $\mathcal{V}(I_w) \subseteq \mathcal{V}(A) \cap \mathcal{V}(B)$, but it is not guaranteed that $I_w \wedge B$ is unsatisfiable. Intuitively, we can think of I_w as being too over-approximated and therefore needing strengthening with respect to B.

Definition 14 (B-adequate). *Let (A, B) be a pair of formulas s.t. $A \wedge B \equiv \bot$ and let I_w be a B_{weak}-interpolant for (A, B). We say that I_w is* B-adequate *iff $I_w \wedge B \equiv \bot$.*

Following the above definition, our purpose is to make a B_{weak}-interpolant I_w B-adequate. We refer to this procedure as *B-Strengthening*.

The purpose of this section is to demonstrate the use of B_{weak}-interpolants for model checking, in particular in the context of ITP.

[2] Some works choose different ways of increasing k. For example, k can be increased by the number of iterations executed in the inner loop: $k = k + n$. In our experiments $k = k + 1$ yielded better results.

Definition 15 (k-n-pair). *Given the formulas $A = I^k_{n-1}(V^0) \wedge \mathrm{TR}(V^0, V^1)$ and $B = path^{1,k} \wedge (\bigvee_{i=1}^{k} \neg p(V^i))$. The pair (A, B) is called a* k-n-pair. *When $A \wedge B \equiv \bot$ we call (A, B) an* inconsistent k-n-pair.

Consider a run of ITP for a given k and n. We aim at computing I^k_n. Let (A, B) be an inconsistent k-n-pair and let I_w be the B_{weak}-interpolant for (A, B). If I_w is B-adequate then it is an interpolant and therefore I^k_n can be defined to be I_w. If I_w is not B-adequate we are required to apply B-Strengthening and transform I_w into an interpolant.

Let us assume that I_w is not B-adequate and that $I_w(V^1) \wedge B$ is satisfiable. There exists a state $s \in I_w$ such that $s(V^1) \wedge B$ is satisfiable. Intuitively, in order to make I_w B-adequate, and by that an interpolant, we would like to remove s from it.

Clearly, $A \wedge s(V^1)$ is unsatisfiable; otherwise $A \wedge B$ would have been satisfiable. Thus, B-Strengthening can be done by iterating all assignments for $I_w(V^1) \wedge B$, extracting a state $s \in I_w$ from an assignment and blocking it in I_w. This is an inefficient way to perform B-Strengthening since the number of such assignments may be too large.

To overcome this, we use knowledge about the problem at hand. Namely, we consider the fact that A is of the following form: $A = I^k_{n-1}(V) \wedge \mathit{TR}(V, V')$.

Definition 16 (Relatively Inductive). *Let R and Q be propositional formulas and M a transition system. We say that Q is* relatively inductive *with respect to R and M if $(R(V) \wedge Q(V)) \wedge TR(V, V') \Rightarrow Q(V')$. When M is clear from the context we omit it.*

Recall that by Def. 13 R^k_n represents an over-approximation of all reachable states after up to n transitions and it is $(k-1)$-adequate (Lemma 1).

Lemma 2. *Let (A, B) be an inconsistent k-n-pair. Let I_w be the B_{weak}-interpolant for (A, B) . If s is an assignment to V s.t. $s(V^1) \wedge B$ is satisfiable, then $R^k_{n-1} \Rightarrow \neg s$ and $R^k_{n-1} \wedge \mathrm{TR} \Rightarrow \neg s'$ hold.*

The above lemma states that if a state s can reach a bad state in up to $k-1$ transitions, it cannot be a state in the set R^k_{n-1}. If we consider a B_{weak}-interpolant derived from the pair (A, B), assuming that $s \in I_w$ (derived from the satisfying assignment to $I_w(V^1) \wedge B$), then s follows the condition in Lemma 2. Therefore, $R^k_{n-1} \Rightarrow \neg s$ and $R^k_{n-1} \wedge \mathit{TR} \Rightarrow \neg s'$ hold and by that $(R^k_{n-1} \wedge \neg s) \wedge \mathit{TR} \Rightarrow \neg s$ holds. By Def. 16 $\neg s$ is relatively inductive with respect to R^k_{n-1}. Therefore, $\neg s$ can be inductively generalized [3].

Inductive generalization results in a sub-clause c of $\neg s$ such that $(R^k_{n-1} \wedge c) \wedge \mathit{TR} \Rightarrow c'$ and $\mathit{INIT} \Rightarrow c$. c can then be used to strengthen I_w and R^k_{n-1}. Adding the clause c to I_w removes s from I_w. This process is then iterated until I_w becomes B-adequate and hence an interpolant. The algorithm for finding the clauses that make I_w B-adequate is described in Fig. 5.

Theorem 1. *Let (A, B) be an inconsistent k-n-pair. Let I_w be a B_{weak}-interpolant and let $c_1, \ldots, c_m$ be clauses s.t.* INIT $\Rightarrow c_i$ *and c_i is relatively inductive w.r.t. R^k_{n-1} for $1 \leq i \leq m$. If $(I_w \wedge \bigwedge_{j=1}^{m} c_j) \wedge B \equiv \bot$ then $I_w \wedge \bigwedge_{j=1}^{m} c_j$ is an interpolant w.r.t $A = (I^k_{n-1}(V^0) \wedge \bigwedge_{j=1}^{m} c_j(V^0)) \wedge \mathrm{TR}(V^0, V^1)$ and $B = path^{1,k} \wedge (\bigvee_{i=1}^{k} \neg p(V^i))$.*

4.3 CNF-ITP: Using B_{weak}-Interpolants in ITP

Above we described how a B_{weak}-interpolant is transformed into an interpolant efficiently for model checking. In this section we present CNF-ITP, a model checking algorithm that is based on ITP. CNF-ITP uses the method described above to compute interpolants. In addition, it uses optimizations that are possible as a result of using interpolants in CNF.

Like the original ITP, our version consists of two nested loops. Since the computation of interpolants is performed in the inner loop, this is where we have made most of our modifications and optimizations. Recall that in the inner loop a BMC formula of a fixed-bound is checked iteratively, where at each iteration only the initial states are replaced by the interpolants computed in a previous iteration. Our modified version of the inner loop appears in Fig. 6

In what follows we consider k to be the unrolling depth used in the inner loop and n to be the iteration during the execution of the inner loop.

The beginning of the loop is similar to the original inner loop of ITP. First, a counterexample of length k is checked (lines: 44-46). If no counterexample exists the pair (A, B) is defined and a B_{weak}-interpolant I_w is computed (line: 50). Then, two optimizations are applied. First, clauses are pushed forward (line: 51). Second, previously computed interpolant is conjoined to the currently computed B_{weak}-interpolant (line: 52). Since I_w may not be B-adequate, the B-Strengthening process may need to add clauses to it (to strengthen it). Adding clauses to I_w before B-Strengthening results in a more efficient B-Strengthening. Moreover, after pushing clauses forward and adding clauses from the previously computed interpolant, I_w may become B-adequate, thereby rendering B-Strengthening redundant.

After applying the two optimizations, B-Strengthening is invoked (line 53). Then the clauses learned during this process are conjoined with R^k_{n-1} and I^k_{n-1} (line 56), and I_w (line 57). After conjoining the clauses, I^k_n is an interpolant. The rest of the loop is identical to the original inner loop of ITP.

We now describe the optimizations in more detail.

Pushing Clauses Forward. Let us consider the interpolant I^k_n computed during the n-th iteration of the inner loop. Since I^k_n is given in CNF, assume that $I^k_{n-1} = \{c_1, \ldots, c_m\}$ where c_i is a clause for every $1 \leq i \leq m$.

```
32: function FindMissingClauses(R,I_w , B, n)
33:     C = ∅
34:     while (I_w ∧ C)(V^1) ∧ B == SAT do      // When C = ∅ it is evaluated as ⊤
35:         Get s ∈ I_w from the SAT assignment
36:         c = InductiveGeneralization(R, s, C)
37:         C = C ∪ c
38:     end while
39:     StoreClauses(n)
40:     return C
41: end function
```

Fig. 5. Find the clauses needed for the B_{weak}-interpolant I_w to be B-adequate

```
42: function ComputeReachableCNF(M,p, k)
43:     R_0^k = INIT, I_0^k = INIT, n = 1
44:     if I_0^k ∧ path^{0,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k)) == SAT then
45:         return cex
46:     end if
47:     repeat
48:         A = I_{n-1}^k(V^0) ∧ TR(V^0, V^1)
49:         B = path^{1,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k))
50:         I_w = GetBWeakInterpolant(A, B)
51:         PushInductiveClauses(I_w, n − 1)
52:         I_w = I_w ∧ I_n^{k−1}      // For k = 1, I_n^0 = ⊤
53:         C = FindMissingClauses(R_{n-1}^k, I_w, B)
54:         I_n^k = I_w
55:         for all c ∈ C do
56:             R_{n-1}^k = R_{n-1}^k ∧ c      // Implicitly conjoining c with I_{n-1}^k
57:             I_n^k = I_n^k ∧ c
58:         end for
59:         if I_n^k ⇒ R_{n-1}^k then
60:             return fixpoint
61:         end if
62:         R_n^k = R_{n-1}^k ∨ I_n^k
63:         n = n + 1
64:     until I_{n-1}^k ∧ path^{0,k} ∧ (¬p(V^1) ∨ ... ∨ ¬p(V^k)) == SAT
65: end function
```

Fig. 6. Inner loop of CNF-ITP

Definition 17. *Let M be a transition system and let $F = \{c_1, \ldots, c_m\}$ be a formula in CNF where c_i is a clause over V for every $1 \leq i \leq m$. A clause c_i for some $1 \leq i \leq n$ is said to be* pushable *if $F(V) \wedge \mathrm{TR}(V, V') \Rightarrow c_i(V')$ holds.*

After the computation of a B_{weak}-interpolant, we try to find pushable clauses. Those clauses can be made part of the new interpolant. Adding the pushable clauses to the B_{weak}-interpolant strengthens it.

Incremental Interpolants. The outer loop of CNF-ITP (and ITP) increases the unrolling depth when a more precise over-approximation is needed. Let I_1^1 be the interpolant computed in the first iteration of the inner loop for $k = 1$ and let I_1^2 be the interpolant computed in the first iteration of the inner loop for $k = 2$. Clearly, since both I_1^1 and I_1^2 over-approximate the states reachable in one transition from the initial states, $I_1^1 \wedge I_1^2$ is also an over-approximation of the same set of states. Usually, the size of the interpolants is an issue. Therefore, whenever the inner loop terminates and the bound is increased, all computed interpolants are discarded and are not re-used [13]. Since our method produces interpolants in CNF that are usually small, this conjunction does not create huge CNF formula. This re-use of previously computed interpolants increases the efficiency of CNF-ITP as compared to ITP.

4.4 CNF-ITP: The Best of ITP and IC3

CNF-ITP uses key elements of ITP and IC3. On the one hand, like ITP, CNF-ITP uses the resolution refutation to get information about the reachable states. This information is only partial, and therefore CNF-ITP also uses *inductive generalization*, a key element of IC3, to complete the computation of reachable states. Since the reachable

states are computed by means of over-approximations, there are cases in which the precision of these approximations must be increased. To do so, CNF-ITP uses unrolling, like in ITP. In addition, it uses the fact that interpolants are given in CNF and tries to reuse clauses that have already been learnt (both by pushing the clauses forward and by using previously computed interpolants). CNF-ITP can be viewed as a hybridization of the monolithic approach (ITP) and the incremental approach (IC3). We believe that there are well-founded grounds for comparing the three algorithms, and that further development can bring about an even tighter integration of ITP and IC3. This discussion, however, is outside the scope of this paper.

5 Experimental Results

Our approach includes two major parts. The first part computes a B_{weak}-interpolant from a resolution refutation, and the second part applies B-Strengthening and a model checking algorithm CNF-ITP. The computation of B_{weak}-interpolants was implemented on top of *MiniSAT 2.2*. CNF-ITP and ITP were implemented in a closed-source model checker. For IC3 we used the publicly available ABC framework[3]. In the results we also include the runtime for ABC's ITP implementation in order to show the efficiency of our implementation.

To evaluate our method we used a representative subset of the HWMCC'12 benchmark set. We chose all valid benchmarks that either ITP or CNF-ITP could prove in the given time frame (56 cases). Table 1 presents 30 out of 56 these cases. All experiments were conducted on a system with an Intel E5-2687W running at 3.1GHz with 32GB of memory. Timeout was set to 900 seconds. As mentioned, we sought to test two aspects: the size of the resulting interpolants and the impact on model checking.

Consider Table 1. Our method generates significantly smaller interpolants[4] in almost every case. Summarizing the average size of all computed interpolants shows that CNF-ITP generates interpolants that are *42 times smaller* than those generated by ITP. Note that average interpolant computation time is nearly the same for both methods.

Another interesting aspect of the comparison between CNF-ITP and ITP is the convergence bound. We can see that in many cases the bound is different. This indicates that the strength of the interpolants computed by the two methods is different and affects the results of the model checking algorithm.

Comparing the run-time of the model checking algorithms shows that our CNF-ITP algorithm outperforms ITP and IC3 in terms of the overall run-time. CNF-ITP outperforms ITP in 21 instances, where in 4 of these instances ITP times out. ITP outperforms CNF-ITP only in 7 cases only. CNF-ITP outperforms IC3 in 11 cases, but IC3 is preferable in 16 cases. CNF-ITP is the absolutely best algorithm in 8 cases.

Analysis of the results in the table shows that whenever the number of clauses in the interpolants computed by CNF-ITP is significantly smaller than the number of clauses in the interpolants computed by ITP, the former performs better.

In the cases where the size of interpolants is fairly the same, ITP performs better. This can be explained by the fact that ITP computes small interpolants when the

[3] `https://bitbucket.org/alanmi/abc`

[4] For ITP, the number of clauses is after translation of the interpolants to CNF.

Table 1. Experiment parameters on part of the benchmarks. *Name*: property name; ♯*Vars*: number of state variables in the cone of influence; k is the bound of the outer loop at which fixpoint was found; total_n is the *total* number of iterations executed by the inner loop; $\text{clauses}_{\text{Avg}}$ is the average number of clauses representing each computed interpolant; Extract[s] is the average time to compute an interpolant in seconds; MC[s] is the total runtime of the algorithm in seconds. Values in boldface are the best of all three. Underlined runtime is for cases where CNF-ITP outperforms ITP and *Italic* is for cases where CNF-ITP outperforms IC3.

		IC3_{ABC}	ITP_{ABC}	ITP					CNF-ITP				
Name	♯Vars	MC[s]	MC[s]	k	total_n	$\text{clauses}_{\text{Avg}}$	Extract[s]	MC[s]	k	total_n	$\text{clauses}_{\text{Avg}}$	Extract[s]	MC[s]
beembkry1b1	76	**4.68**	758	15	72	94495	3.14	792	20	83	**1830**	0.65	248
beemcoil1b1	132	**11.77**	TO	9	51	45563	2.09	577	11	52	**487**	0.28	201
beemexit5f1	246	**7.11**	106	25	218	15792	1.21	611	25	255	**792**	0.3	466
beemfish4f1	94	**4.41**	TO	15	50	63423	3.26	TO	14	59	**1348**	0.72	85
beemfwt5f2	3045	543.32	14.68	5	9	854	0.009	2.2	5	10	**22**	0.00	**2.01**
beemfwt1b1	1214	321.86	396	4	18	5721	0.23	**10.58**	4	15	**62**	38.14	TO
beemndhm2f2	251	13.53	138	7	49	29051	1.07	213	5	10	**143**	0.14	**2.92**
beempgmprot1b1	1025	**8.03**	97	33	218	1121	0.82	TO	19	113	**61**	0.00	27.65
beempgmprot7b1	1033	591.49	204	27	168	2717	1.23	TO	18	153	**237**	0.00	**36.21**
beemtlphn5f1	249	29.81	TO	12	60	73460	5.27	TO	20	66	**1197**	0.06	TO
beemtrngt2b1	170	**1.55**	TO	29	193	31942	1.95	TO	15	154	**618**	0.08	23.42
beemtrngt4b1	228	44.71	TO	29	196	22144	1.56	TO	30	281	**371**	0.71	TO
bob05	2404	**7.5**	275	24	121	962	0.37	221	24	136	**198**	0.38	113
bob1u05cu	4377	**7.66**	235	24	124	3116	0.45	251	24	147	**272**	0.19	152
eijkbs3330	246	**7.2**	TO	3	6	764550	22.74	TO	3	9	**5873**	10.44	154
6s38	1931	TO	TO	10	33	84988	1.19	TO	7	22	**4299**	15.7	**423**
6s108	782	**4.83**	TO	8	43	89493	3.67	TO	7	25	**1787**	0.56	42.65
6s120	58	**0.71**	4.1	3	6	**365**	0.34	6.5	3	8	373	0.12	2.92
6s121	419	821.54	TO	24	214	4542	0.08	46.25	18	98	**389**	0.04	**14.42**
6s132	139	**2.87**	7.5	7	13	35973	2.88	85	5	13	**4221**	3.5	76
6s136	3342	TO	3.1	20	58	2471	0.005	4.4	20	53	**16**	0.00	**1.94**
6s151	150	TO	TO	14	515	**1998**	0.22	**461**	10	122	6226	3.15	TO
6s159	252	0.03	7.8	15	143	656	0.01	4.9	10	60	**27**	0.00	0.34
6s164	198	8.96	TO	18	77	753	0.02	3.7	18	85	**135**	0.006	**2.43**
6s181	607	TO	26.2	8	19	63509	4.58	**232**	6	10	**2556**	6.45	TO
intel021	365	TO	TO	18	316	2503	0.05	**51.3**	18	489	**331**	0.14	*117*
intel022	530	TO	TO	21	435	18818	0.5	**629**	20	555	**585**	1.05	TO
intel024	357	TO	TO	15	233	3087	0.04	**34.5**	15	341	**206**	0.12	*83*
intel031	531	TO	TO	21	268	3465	0.15	114	18	235	**183**	0.03	**61**
intel034	3297	TO	TO	16	425	1477	0.02	**85**	16	432	**83**	0.19	*119*
Total		10544	16672			1469009	59	12535			34928	83	**7854**

resolution refutation is small. Therefore, computing the interpolants in ITP is more efficient in these cases since it only requires linear traversal over the resolution refutation. In contrast, our method requires B-Strengthening, a process that is in some cases expensive. We conclude that when the resulting interpolants in ITP are large, CNF-ITP has a significant advantage in the vast majority of cases.

6 Conclusion

We have presented a novel approach for deriving interpolants for SAT-based model checking. Our procedure generates small interpolants in CNF using a twofold scheme: First, an interpolant approximation is computed with an algorithm that exploits resolution refutation properties. Following this, inductive reasoning is used to complete transforming the approximation into an interpolant. Our experiments show that our approach generates interpolants that are much smaller (by 42 times overall) than those generated by the classical ITP approach.

In addition, we have implemented CNF-ITP, a model checking algorithm that uses the above method to compute interpolants. CNF-ITP significantly outperformed ITP and outperformed IC3 in a large number of cases. We believe that this approach may be further developed and enhanced, yielding an even more efficient model checking algorithm.

Acknowledgments. The authors would like to thank Håkan Hjort and Paul Inbar for their valuable comments.

References

1. Bar-Ilan, O., Fuhrmann, O., Hoory, S., Shacham, O., Strichman, O.: Linear-time reductions of resolution proofs. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 114–128. Springer, Heidelberg (2009)
2. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Advances in Computers 58, 118–149 (2003)
3. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
4. Cabodi, G., Murciano, M., Nocco, S., Quer, S.: Stepping forward with interpolants in unbounded model checking. In: ICCAD, pp. 772–778 (2006)
5. Chockler, H., Ivrii, A., Matsliah, A.: Computing interpolants without proofs. In: HVC (2012)
6. Craig, W.: Linear reasoning. a new form of the herbrand-gentzen theorem. J. Symb. Log. 22(3) (1957)
7. Davis, M., Putnam, H.: A computing procedure for quantification theory. J. ACM 7(3), 201–215 (1960)
8. D'Silva, V., Kroening, D., Purandare, M., Weissenbacher, G.: Interpolant strength. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 129–145. Springer, Heidelberg (2010)
9. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005)
10. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: DATE, pp. 886–891 (2003)
11. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
12. Jhala, R., McMillan, K.L.: Interpolant-based transition relation approximation. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 39–51. Springer, Heidelberg (2005)
13. Marques-Silva, J.: Interpolant learning and reuse in SAT-based model checking. Electr. Notes Theor. Comput. Sci. 174(3), 31–43 (2007)
14. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
15. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
16. McMillan, K.L., Amla, N.: Automatic Abstraction without Counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
17. Pudlák, P.: Lower bounds for resolution and cutting plane proofs and monotone computations. J. Symb. Log. 62(3) (1997)

18. Rollini, S.F., Bruttomesso, R., Sharygina, N.: An efficient and flexible approach to resolution proof reduction. In: Raz, O. (ed.) HVC 2010. LNCS, vol. 6504, pp. 182–196. Springer, Heidelberg (2010)
19. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
20. Vizel, Y., Grumberg, O.: Interpolation-sequence based model checking. In: FMCAD (2009)
21. Vizel, Y., Grumberg, O., Shoham, S.: Intertwined forward-backward reachability analysis using interpolants. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 308–323. Springer, Heidelberg (2013)
22. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable Boolean formula. In: SAT (2003)

Disjunctive Interpolants for Horn-Clause Verification

Philipp Rümmer[1], Hossein Hojjat[2], and Viktor Kuncak[2]

[1] Uppsala University, Sweden
[2] Swiss Federal Institute of Technology Lausanne (EPFL)

Abstract. One of the main challenges in software verification is efficient and precise compositional analysis of programs with procedures and loops. Interpolation methods remains one of the most promising techniques for such verification, and are closely related to solving Horn clause constraints. We introduce a new notion of interpolation, disjunctive interpolation, which solves a more general class of problems in one step compared to previous notions of interpolants, such as tree interpolants or inductive sequences of interpolants. We present algorithms and complexity for construction of disjunctive interpolants, as well as their use within an abstraction-refinement loop. We have implemented Horn clause verification algorithms that use disjunctive interpolants and evaluate them on benchmarks expressed as Horn clauses over the theory of integer linear arithmetic.

1 Introduction

Software model checking has greatly benefited from the combination of a number of seminal ideas: automated abstraction through theorem proving [8], exploration of finite-state abstractions, and counterexample-driven refinement [3]. Even though these techniques can be viewed independently, the effectiveness of verification has been consistently improving by providing more sophisticated communication between these steps. Often, carefully chosen search aspects are being pushed into a learning-enabled constraint solver, resulting in better overall verification performance. An essential advance was to use interpolants derived from unsatisfiability proofs to refine the abstraction [13]. In recent years, we have seen significant progress in interpolating methods for different logical constraints [4, 5, 21], and a wealth of more general forms of interpolation [1, 12, 21, 24]. In this paper we identify a new notion, *disjunctive interpolants*, which are more general than tree interpolants and inductive sequences of interpolants. Like tree interpolation [12, 21], a disjunctive interpolation query is a tree-shaped constraint specifying the interpolants to be derived; however, in disjunctive interpolation, branching in the tree can represent both conjunctions and disjunctions. We present an algorithm for solving the interpolation problem, relating it to a subclass of recursion-free Horn clauses [10, 22, 23]. We then consider solving general recursion-free Horn clauses and show that this problem is solvable whenever the logic admits interpolation. We establish tight complexity bounds for solving recursion-free Horn clauses for propositional logic (PSPACE) and for integer linear arithmetic (co-NEXPTIME). In contrast, the disjunctive interpolation problem remains in coNP for these logics. We also show how to use solvers for recursion-free Horn clauses to verify recursive Horn clauses using counterexample-driven predicate abstraction. We present an algorithm and experimental results on publicly available benchmarks.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 347–363, 2013.

1.1 Related Work

There is a long line of research on Craig **interpolation** methods, and generalised forms of interpolation tailored to verification. For an overview of interpolation in the presence of theories, we refer the reader to [4, 5]. Binary Craig interpolation for implications $A \rightarrow C$ goes back to [6], was used on conjunctions $A \wedge B$ in [19], and generalised to inductive sequences of interpolants in [13, 20]. The concept of tree interpolation, strictly generalising inductive sequences of interpolants, is presented in the documentation of the interpolation engine iZ3 and in [21]; the computation of tree interpolants by computing a sequence of binary interpolants is also described in [12]. In this paper, we present a new form of interpolation, *disjunctive interpolation*, which is strictly more general than sequences of interpolants and tree interpolants. Our implementation supports Presburger arithmetic, including divisibility constraints [4], which is rarely supported by existing tools, yet helpful in practice [15].

A further generalisation of inductive sequences of interpolants are restricted DAG interpolants [1], which also include disjunctiveness in the sense that multiple paths through a program can be handled simultaneously. Disjunctive interpolants are incomparable in power to restricted DAG interpolants, since the former does not handle interpolation problems in the form of DAGs, while the latter does not subsume tree interpolation. A combination of the two kinds of interpolants ("disjunctive DAG interpolation") is strictly more powerful (and harder) than disjunctive interpolation, see Sect. 5.1 for a complexity-theoretic analysis. We discuss techniques and heuristics to practically handle shared sub-trees in disjunctive interpolation, extending the benefits of DAG interpolation to recursive programs.

Inter-procedural **software model checking** with interpolants has been an active area of research. In the context of predicate abstraction, it has been discussed how well-scoped invariants can be inferred [13] in the presence of function calls. Based on the concept of Horn clauses, a predicate abstraction-based algorithm for bottom-up construction of function summaries was presented in [9]. Encoding into Horn clauses is also used in logic programming community [23]. Verification of programs with procedures is described in [12] (using nested word automata) as well as in [2]. Function summaries generated using interpolants have also been used in bounded model checking [26]. Researchers also showed how to lift these techniques to higher-order programs [17, 28].

The use of **Horn clauses** as intermediate representation for verification was proposed in [10], with the verification of concurrent programs as main application. The underlying procedure for solving sets of recursion-free Horn clauses, over the combined theory of linear *rational* arithmetic and uninterpreted functions, was presented in [11]. An algorithm to solve recursion-free systems of Horn constraints by repeated computation of binary interpolants was given in [27], for the purpose of type inference. A range of further applications of Horn clauses, including inter-procedural model checking, was given in [9]. Horn clauses are also used as a format for verification problems supported by the SMT solver Z3 [14]. Our paper extends this direction by presenting general results about solvability and computational complexity, independent of any particular calculus. Our experiments are with linear *integer* arithmetic, arguably a more faithful model of discrete computation than rationals [15].

2 Example: Verification of Recursive Predicates

We start by showing how our approach can verify programs encoded as Horn clauses, by means of predicate abstraction and a theorem prover for Presburger arithmetic. Fig. 1 shows an example of a system of Horn clauses that compute the greatest common divisor of its first and its second argument in its third argument. After invoking the gcd operation on the equal positive numbers M and N, we wish to check whether it is possible for the result R to be more than the M. In general, we encode error conditions as Horn clauses with *false* in their head, and refer to such clauses as error clauses, although such clauses do not have a special semantic status in our system. When executed with these clauses as input, our verification tool automatically identifies that the definition of gcd(M,N,R) as the predicate $(M = N) \rightarrow (M \geq R)$ gives a solution to these Horn clauses. In terms of safety (partial correctness), this means that the error condition cannot be reached.

Our approach uses counterexample-driven refinement to perform verification. In this example, the abstraction of Horn clauses starts with a trivial set of predicates, containing only the predicate *false*, which is assumed to be a valid approximation until proven otherwise. Upon examining a clause that has a concrete satisfiable formula on the right-hand side (e.g. $M = N \wedge R = M$), we rule out *false* as the approximation of gcd. In the absence of other candidate predicates, the approximation of gcd becomes the conjunction of an empty set of predicates, which is *true*. Using this approximation the error clause is no longer satisfied. At this point the algorithm checks whether a true error is reached by directly chaining the clauses involved in computing the approximation of predicates. This amounts to checking whether the following recursion-free subset of clauses has a solution:

(1) gcd(M,N,R) $\leftarrow$ M = N $\wedge$ R = M
(4) false $\leftarrow$ M $\geq$ 0 $\wedge$ M = N $\wedge$ gcd(M,N,R) $\wedge$ R > M

The solution to above problem is any formula $I(M, N, R)$ such that

(1) gcd(M,N,R) $\leftarrow$ M = N $\wedge$ R = M
(2) gcd(M,N,R) $\leftarrow$ M > N $\wedge$ M1 = M − N $\wedge$ gcd(M1,N,R)
(3) gcd(M,N,R) $\leftarrow$ M < N $\wedge$ N1 = N − M $\wedge$ gcd(M,N1,R)
(4) false $\leftarrow$ M $\geq$ 0 $\wedge$ M = N $\wedge$ gcd(M,N,R) $\wedge$ R > M

Fig. 1. Horn clauses computing the greatest common divisor of two numbers and an assertion on result. Variables are universally quantified in each clause.

(1) gcd(M,N,R) $\leftarrow$ M = N $\wedge$ R = M
(1') gcd1(M,N,R) $\leftarrow$ M = N $\wedge$ R = M
(2') gcd(M,N,R) $\leftarrow$ M > N $\wedge$ M1 = M − N $\wedge$ gcd1(M1,N,R)
(3') gcd(M,N,R) $\leftarrow$ M < N $\wedge$ N1 = N − M $\wedge$ gcd1(M,N1,R)
(4) false $\leftarrow$ M $\geq$ 0 $\wedge$ M = N $\wedge$ gcd(M,N,R) $\wedge$ R > M

Fig. 2. Extended recursion-free approximation of the Horn clauses in Fig. 1

I(M,N,R) ← M = N ∧ R = M
false ← M ≥ 0 ∧ M = N ∧ I(M,N,R) ∧ R > M

This is precisely an interpolant of $M = N \wedge R = M$ and $M \geq 0 \wedge M = N \wedge R > M$. A valid interpolant is $P_1(M, N, R) \equiv M \geq R$. Choosing this interpolant eliminates the current contradiction for Horn clauses and P_1 is added into a list of abstraction predicates for the relation gcd. Because the predicates approximating gcd are now updated, we consider the abstraction of the system in terms of these predicates.

The predicate P_1 is not a conjunct in a valid approximation for gcd in clause (2), so the following recursion-free unfolding is not solved by the approximation so far:

(1) gcd(M,N,R) ← M = N ∧ R = M
(2') gcd1(M,N,R) ← M > N ∧ M1 = M − N ∧ gcd(M1,N,R)
(4') false ← M ≥ 0 ∧ M = N ∧ gcd1(M,N,R) ∧ R > M

This particular problem could be reduced to solving an interpolation sequence, but it is more natural to think of it simply as a solution for recursion-free Horn clauses. A solution is an interpretation of the relations gcd and gcd1 as ternary relations on integers, such that the clauses are true. Note that this problem could also be viewed as the computation of tree interpolants, which are also a special case of solving recursion-free Horn clauses, as are DAG interpolants and a new notion of disjunctive tree interpolants that we introduce. In line with [9–11] we observe that recursion-free clauses are a perfect fit for counterexample-driven verification: they allow us to provide the theorem proving procedure with much more information that they can use to refine abstractions. In the limit, the original set of clauses or its recursive unfoldings are its own approximations, some of them exact, but the advantage of *recursion-free* Horn clauses is that their solvability is decidable under very general conditions. This provides us with a solid theorem proving building block to construct robust and predictable solvers for the undecidable recursive case. Our paper describes a new such building block: disjunctive interpolants, which correspond to a subclass of non-recursive Horn clauses.

To illustrate disjunctive interpolants, Fig. 2 provides another recursion-free approximations of the problem. In this approximation we can distinguish 3 different paths from the error clause (4) through the clauses (1'), (2') and (3') to ground formulae. The traditional refinement approach using e.g. tree interpolation typically removes the 3 instances of the spurious counter-examples using 3 interpolation calls. A novelty of disjunctive interpolation is removing the different choices of counter-examples altogether using a single call to the interpolating theorem prover. Eliminating more counter-examples at once can reduce the number of iterations and increase convergence.

3 Formulae and Horn Clauses

Constraint languages. Throughout this paper, we assume that a first-order vocabulary of *interpreted symbols* has been fixed, consisting of a set $\mathcal{F}$ of fixed-arity function symbols, and a set $\mathcal{P}$ of fixed-arity predicate symbols. Interpretation of $\mathcal{F}$ and $\mathcal{P}$ is determined by a class $\mathcal{S}$ of structures (U, I) consisting of non-empty universe U, and a mapping I that assigns to each function in $\mathcal{F}$ a set-theoretic function over U, and to each predicate in $\mathcal{P}$ a set-theoretic relation over U. As a convention, we assume

the presence of an equation symbol "=" in $\mathcal{P}$, with the usual interpretation. Given a countably infinite set $\mathcal{X}$ of variables, a *constraint language* is a set *Constr* of first-order formulae over $\mathcal{F}, \mathcal{P}, \mathcal{X}$ For example, the language of quantifier-free Presburger arithmetic has $\mathcal{F} = \{+, -, 0, 1, 2, \ldots\}$ and $\mathcal{P} = \{=, \leq, |\}$).

A constraint is called *satisfiable* if it holds for some structure in $\mathcal{S}$ and some assignment of the variables $\mathcal{X}$, otherwise *unsatisfiable*. We say that a set $\Gamma \subseteq$ *Constr* of constraints *entails* a constraint $\phi \in$ *Constr* if every structure and variable assignment that satisfies all constraints in Γ also satisfies ϕ; this is denoted by $\Gamma \models \phi$.

$fv(\phi)$ denotes the set of free variables in constraint ϕ. We write $\phi[x_1, \ldots, x_n]$ to state that a constraint contains (only) the free variables $x_1, \ldots, x_n$, and $\phi[t_1, \ldots, t_n]$ for the result of substituting the terms $t_1, \ldots, t_n$ for $x_1, \ldots, x_n$. Given a constraint ϕ containing the free variables $x_1, \ldots, x_n$, we write $Cl_\forall(\phi)$ for the *universal closure* $\forall x_1, \ldots, x_n.\phi$.

Positions. We denote the set of *positions* in a constraint ϕ by *positions*(ϕ). For instance, the constraint $a \wedge \neg a$ has 4 positions, corresponding to the sub-formulae $a \wedge \neg a$, $\neg a$, and the two occurrences of a. The sub-formula of a formula ϕ underneath a position p is denoted by $\phi \downarrow p$, and we write $\phi[p/\psi]$ for the result of replacing the sub-formula $\phi \downarrow p$ with ψ. Further, we write $p \leq q$ if position p is above q (that is, q denotes a position within the sub-formula $\phi \downarrow p$), and $p < q$ if p is strictly above q.

Craig interpolation is the main technique used to construct and refine abstractions in software model checking. A binary interpolation problem is a conjunction $A \wedge B$ of constraints. A *Craig interpolant* is a constraint I such that $A \models I$ and $B \models \neg I$, and such that $fv(I) \subseteq fv(A) \cap fv(B)$. The existence of an interpolant implies that $A \wedge B$ is unsatisfiable. We say that a constraint language has the *interpolation property* if also the opposite holds: whenever $A \wedge B$ is unsatisfiable, there is an interpolant I.

3.1 Horn Clauses

To define the concept of Horn clauses, we fix a set $\mathcal{R}$ of uninterpreted fixed-arity *relation symbols,* disjoint from $\mathcal{P}$ and $\mathcal{F}$. A *Horn clause* is a formula $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H$ where

- C is a constraint over $\mathcal{F}, \mathcal{P}, \mathcal{X}$;
- each B_i is an application $p(t_1, \ldots, t_k)$ of a relation symbol $p \in \mathcal{R}$ to first-order terms over $\mathcal{F}, \mathcal{X}$;
- H is similarly either an application $p(t_1, \ldots, t_k)$ of $p \in \mathcal{R}$ to first-order terms, or is the constraint *false*.

H is called the *head* of the clause, $C \wedge B_1 \wedge \cdots \wedge B_n$ the *body.* In case $C = true$, we usually leave out C and just write $B_1 \wedge \cdots \wedge B_n \rightarrow H$. First-order variables (from $\mathcal{X}$) in a clause are considered implicitly universally quantified; relation symbols represent set-theoretic relations over the universe U of a structure $(U, I) \in \mathcal{S}$. Notions like (un)satisfiability and entailment generalise straightforwardly to formulae with relation symbols.

A *relation symbol assignment* is a mapping $sol : \mathcal{R} \rightarrow$ *Constr* that maps each n-ary relation symbol $p \in \mathcal{R}$ to a constraint $sol(p) = C_p[x_1, \ldots, x_n]$ with n free variables. The *instantiation* $sol(h)$ of a Horn clause h is defined by:

$$sol(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow p(\bar{t})) \;=\; C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow sol(p)[\bar{t}]$$
$$sol(C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow \mathit{false}) \;=\; C \wedge sol(p_1)[\bar{t}_1] \wedge \cdots \wedge sol(p_n)[\bar{t}_n] \rightarrow \mathit{false}$$

Definition 1 (Solvability). *Let $\mathcal{HC}$ be a set of Horn clauses over relation symbols $\mathcal{R}$.*

1. *$\mathcal{HC}$ is called semantically solvable if for every structure $(U, I) \in \mathcal{S}$ there is an interpretation of the relation symbols $\mathcal{R}$ as set-theoretic relations over U such that the universally quantified closure $Cl_\forall(h)$ of every clause $h \in \mathcal{HC}$ holds in (U, I).*
2. *$\mathcal{HC}$ is called syntactically solvable if there is a relation symbol assignment sol such that for every structure $(U, I) \in \mathcal{S}$ and every clause $h \in \mathcal{HC}$ it is the case that $Cl_\forall(sol(h))$ is satisfied.*

Note that, in the special case when $\mathcal{S}$ contains only one structure, $\mathcal{S} = \{(U, I)\}$, semantic solvability reduces to the existence of relations interpreting $\mathcal{R}$ that extend the structure (U, I) in such a way to make all clauses true. In other words, Horn clauses are solvable in a structure if and only if the extension of the theory of (U, I) by relation symbols $\mathcal{R}$ in the vocabulary and by given Horn clauses as axioms is consistent.

Clearly, if a set of Horn clauses is syntactically solvable, then it is also semantically solvable. The converse is not true in general, because the solution need not be expressible in the constraint language (see Appendix E of [25] for an example).

A set $\mathcal{HC}$ of Horn clauses induces a *dependence relation* $\rightarrow_{\mathcal{HC}}$ on $\mathcal{R}$, defining $p \rightarrow_{\mathcal{HC}} q$ if there is a Horn clause in $\mathcal{HC}$ that contains p in its head, and q in the body. The set $\mathcal{HC}$ is called *recursion-free* if $\rightarrow_{\mathcal{HC}}$ is acyclic, and *recursive* otherwise. In the next sections we study the solvability problem for recursion-free Horn clauses; in particular, Theorem 2 below characterises the relationship between syntactic and semantic solvability for recursion-free Horn clauses. This case is relevant, since solvers for recursion-free Horn clauses form a main component of many general Horn-clause-based verification systems [9, 10].

4 Disjunctive Interpolants and Body-Disjoint Horn Clauses

Having defined the classical notions of interpolation and Horn clauses, we now present our notion of disjunctive interpolants, and the corresponding class of Horn clauses. Our inspiration are generalized forms of Craig interpolation, such as inductive sequences of interpolants [13, 20] or tree interpolants [12, 21]. We introduce disjunctive interpolation as a new form of interpolation that is tailored to the refinement of abstractions in Horn clause verification, strictly generalising both inductive sequences of interpolants and tree interpolation. Disjunctive interpolation problems can specify both conjunctive and disjunctive relationships between interpolants, and are thus applicable for simultaneous analysis of multiple paths in a program, but also tailored to inter-procedural analysis or verification of concurrent programs [9].

Disjunctive interpolation problems correspond to a specific fragment of recursion-free Horn clauses, namely recursion-free body-disjoint Horn clauses (see Sect. 4.1). The definition of disjunctive interpolation is chosen deliberately to be as general as possible, while still avoiding the high computational complexity of solving general systems of recursion-free Horn clauses. Computational complexity is discussed in Sect. 5.1.

We introduce disjunctive interpolants as a form of *sub-formula abstraction*. For example, given an unsatisfiable constraint $\phi[\alpha]$ containing α as a sub-formula in a positive position, the goal is to find an abstraction α' such that $\alpha \models \alpha'$ and $\alpha[\alpha'] \models \mathit{false}$, and such that α' only contains variables common to α and $\phi[\mathit{true}]$. Generalizing this to any number of subformulas, we obtain the following.

Definition 2 (Disjunctive interpolant). *Let ϕ be a constraint, and $pos \subseteq positions(\phi)$ a set of positions in ϕ that are only underneath the connectives $\wedge$ and $\vee$. A disjunctive interpolant is a map $I : pos \to Constr$ from positions to constraints such that:*

1. *For each position $p \in pos$, with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid p < q \text{ and } \neg\exists r \in pos.\ p < r < q\}$ we have*
$$(\phi[q_1/I(q_1), \ldots, q_n/I(q_n)])\downarrow p \models I(p)\,,$$
2. *For the topmost positions $\{q_1, \ldots, q_n\} = \{q \in pos \mid \neg\exists r \in pos.\ r < q\}$ we have*
$$\phi[q_1/I(q_1), \ldots, q_n/I(q_n)] \models \mathit{false}\,,$$
3. *For each position $p \in pos$, we have $fv(I(p)) \subseteq fv(\phi\downarrow p) \cap fv(\phi[p/\mathit{true}])$.*

Example 1. Consider $A_p \wedge B$, with position p pointing to the sub-formula A, and $pos = \{p\}$. The disjunctive interpolants for $A \wedge B$ and pos coincide with the ordinary binary interpolants for $A \wedge B$.

Example 2. Consider the formula $\phi = (\cdots(((T_1)_{p_1} \wedge T_2)_{p_2} \wedge T_3)_{p_3} \wedge \cdots)_{p_{n-1}} \wedge T_n$ and positions $pos = \{p_1, \ldots, p_{n-1}\}$. Disjunctive interpolants for ϕ and pos correspond to inductive sequences of interpolants [13, 20]. Note that we have the entailments $T_1 \models I(p_1),\ I(p_1) \wedge T_2 \models I(p_2),\ \ldots,\ I(p_{n-1}) \wedge T_n \models \mathit{false}$.

Example 3. Tree interpolation problems correspond to disjunctive interpolation with a set pos of positions that are only underneath $\wedge$ (and never underneath $\vee$). We give a precise definition and results about the existence of tree interpolants in [24].

Example 4. We consider the example given in Fig. 2, Sect. 2. To compute a solution for the Horn clauses, we first *expand* the Horn clauses into a constraint, by means of exhaustive inlining/resolution (see Sect. 5), obtaining a disjunctive interpolation problem:

$$\begin{aligned}
\mathit{false} &\rightsquigarrow M \geq 0 \wedge M = N \wedge \mathsf{gcd}(M, N, R) \wedge R > M \\
&\rightsquigarrow \begin{pmatrix} M \geq 0 \\ \wedge\ M = N \\ \wedge\ R > M \end{pmatrix} \wedge \begin{pmatrix} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge \mathsf{gcd1}(M_1, N, R) \\ \vee \\ M < N \wedge N_1 = N - M \wedge \mathsf{gcd1}(M, N_1, R) \end{pmatrix} \\
&\rightsquigarrow \begin{pmatrix} M \geq 0 \\ \wedge\ M = N \\ \wedge\ R > M \end{pmatrix} \wedge \begin{pmatrix} M = N \wedge R = M \\ \vee \\ M > N \wedge M_1 = M - N \wedge (M_1 = N \wedge R = M_1)_q \\ \vee \\ M < N \wedge N_1 = N - M \wedge (M = N_1 \wedge R = M)_r \end{pmatrix}_p
\end{aligned}$$

In the last formula, the positions p, q, r corresponding to the relation symbol gcd and the two occurrences of gcd1 are marked. It can be observed that the last formula is unsatisfiable, and that $I = \{p \mapsto ((M = N) \rightarrow (M \geq R)), q \mapsto \mathit{true}, r \mapsto \mathit{true}\}$ is a disjunctive interpolant. A solution for the Horn clauses can be derived from the interpolant by conjoining the constraints derived for the two occurrences of gcd1:

$$\mathsf{gcd}(M, N, R) \;=\; ((M = N) \rightarrow (M \geq R)), \quad \mathsf{gcd1}(M, N, R) \;=\; \mathit{true}$$

Theorem 1. *Suppose ϕ is a constraint, and suppose $pos \subseteq positions(\phi)$ is a set of positions in ϕ that are only underneath the connectives $\wedge$ and $\vee$. If Constr is a constraint language that has the interpolation property, then a disjunctive interpolant I exists for ϕ and pos if and only if ϕ is unsatisfiable.*

Proof. "$\Rightarrow$" By means of simple induction, we can derive that $\phi \downarrow p \models I(p)$ holds for every disjunctive interpolant I for ϕ and *pos*, and for every $p \in pos$. From Def. 2, it then follows that ϕ is unsatisfiable.

"$\Leftarrow$" Suppose ϕ is unsatisfiable. We encode the disjunctive interpolation problem into a (conjunctive) tree interpolation problem (following the terminology in [24]) by adding auxiliary Boolean variables.[1] Wlog, we assume that *pos* contains the root position *root* of ϕ. The graph of the tree interpolation problem is (pos, E), with the edge relation $E = \{(p, q) \mid p < q \text{ and } \neg\exists r.\, p < r < q\}$. For every $p \in pos$, let a_p be a fresh Boolean variable. We label the nodes of the tree using the function $\phi_L : pos \rightarrow Constr$. For each position $p \in pos$, with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid E(p, q)\}$ we define

$$\phi_L(p) \;=\; \begin{cases} \phi[q_1/a_{q_1}, \ldots, q_n/a_{q_n}] & \text{if } p = \mathit{root} \\ \neg a_p \vee (\phi[q_1/a_{q_1}, \ldots, q_n/a_{q_n}])\downarrow p & \text{otherwise} \end{cases}$$

Observe that $\bigwedge_{p \in pos} \phi_L(p)$ is unsatisfiable. According to [24], a tree interpolant I_T exists for this labelling function. By construction, for non-root positions $p \in pos \setminus \{root\}$ the interpolant labelling is equivalent to $I_T(p) \equiv \neg a_p \vee I_p$, where I_p does not contain any further auxiliary Boolean variables. We can then construct a disjunctive interpolant I for the original problem as

$$I(p) \;=\; \begin{cases} \mathit{false} & \text{if } p = \mathit{root} \\ I_p & \text{otherwise} \end{cases}$$

To see that I is a disjunctive interpolant, observe that for each position $p \in pos$ with direct children $\{q_1, \ldots, q_n\} = \{q \in pos \mid E(p, q)\}$ the following entailment holds (since I_T is a tree interpolant): $\phi_L(p) \wedge (\neg a_{q_1} \vee I_{q_1}) \wedge \cdots \wedge (\neg a_{q_n} \vee I_{q_n}) \models I_T(p)$
Via Boolean reasoning this implies: $(\phi[q_1/I_{q_1}, \ldots, q_n/I_{q_n}])\downarrow p \models I(p)$. □

The proof provides a constructive method to solve disjunctive interpolation problems, by means of transformation to a tree interpolation problem. This is also the algorithm that we used in our experiments in Sect. 6.2; practical aspects of this approach are discussed in the beginning of Sect. 6.

[1] The concept of auxiliary Boolean variables to represent interpolation problems has also been used in [26] and [2], for the purpose of extracting function summaries in model checking.

4.1 Solvability of Body-Disjoint Horn Clauses

The relationship between Craig interpolation and (syntactic) solutions of Horn clauses has been observed in [11]. Disjunctive interpolation corresponds to a specific class of recursion-free Horn clauses, namely Horn clauses that are *body disjoint:*

Definition 3. *A finite, recursion-free set $\mathcal{HC}$ of Horn clauses is body disjoint if for each relation symbol p there is at most one clause containing p in its body, and every clause contains p at most once.*

An example for body-disjoint clauses is the subset $\{(1), (4)\}$ of clauses in Fig. 1. Syntactic solutions of a set $\mathcal{HC}$ of body-disjoint Horn clauses can be computed by solving a disjunctive interpolation problem; vice versa, every disjunctive interpolation problem can be translated into an equivalent set of body-disjoint clauses.

In order to extract an interpolation problem from $\mathcal{HC}$, we first normalise the clauses: for every relation symbol $p \in \mathcal{R}$, we fix a unique vector of variables $\bar{x}_p$, and rewrite $\mathcal{HC}$ such that p only occurs in the form $p(\bar{x}_p)$. This is possible due to the fact that $\mathcal{HC}$ is body disjoint. The translation from Horn clauses to a disjunctive interpolation problem is done recursively, similar in spirit to inlining of function invocations in a program; thanks to body-disjointness, the encoding is polynomial.

$$enc(\mathcal{HC}) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to false) \in \mathcal{HC}} C \wedge enc'(B_1) \wedge \cdots \wedge enc'(B_n)$$

$$enc'(p(\bar{x}_p)) = \left(\bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \to p(\bar{x}_p)) \in \mathcal{HC}} C \wedge enc'(B_1) \wedge \cdots \wedge enc'(B_n) \right)_{l_p}$$

Note that the resulting formula $enc(\mathcal{HC})$ contains a unique position l_p at which the definition of a relation symbol p is inlined; in the second equation, this position is marked with l_p. Any disjunctive interpolant I for this set of positions represents a syntactic solution of $\mathcal{HC}$, and vice versa.

5 Solvability of Recursion-Free Horn Clauses

The previous section discussed how the class of recursion-free body-disjoint Horn clauses can be solved by reduction to disjunctive interpolation. We next show that this construction can be generalised to arbitrary systems of recursion-free Horn clauses. In absence of the body-disjointness condition, however, the encoding of Horn clauses as interpolation problems can incur a potentially exponential blowup. We give a complexity-theoretic argument justifying that this blowup cannot be avoided in general. This puts disjunctive interpolation (and, equivalently, body-disjoint Horn clauses) at a sweet spot: preserving the relatively low complexity of ordinary binary Craig interpolation, while carrying much of the flexibility of the Horn clause framework.

We first introduce the exhaustive *expansion* $exp(\mathcal{HC})$ of a set $\mathcal{HC}$ of Horn clauses, which generalises the Horn clause encoding from the previous section. We write $C' \wedge B'_1 \wedge \cdots \wedge B'_n \to H'$ for a fresh variant of a Horn clause $C \wedge B_1 \wedge \cdots \wedge B_n \to H$,

i.e., the clause obtained by replacing all free first-order variables with fresh variables. Expansion is then defined by the following recursive functions:

$$exp(\mathcal{HC}) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow false) \in \mathcal{HC}} C' \wedge exp'(B'_1) \wedge \cdots \wedge exp'(B'_n)$$

$$exp'(p(\bar{t})) = \bigvee_{(C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow p(\bar{s})) \in \mathcal{HC}} C' \wedge exp'(B'_1) \wedge \cdots \wedge exp'(B'_n) \wedge \bar{t} = \bar{s}'$$

Note that *exp* is only well-defined for finite and recursion-free sets of Horn clauses, since the expansion might not terminate otherwise.

Theorem 2 (Solvability of recursion-free Horn clauses). *Let $\mathcal{HC}$ be a finite, recursion-free set of Horn clauses. If the underlying constraint language has the interpolation property, then the following statements are equivalent:*

1. *$\mathcal{HC}$ is semantically solvable;*
2. *$\mathcal{HC}$ is syntactically solvable;*
3. *$exp(\mathcal{HC})$ is unsatisfiable.*

Proof. $2 \Rightarrow 1$ holds because a syntactic solution gives rise to a semantic solution by interpreting the solution constraints. $\neg 3 \Rightarrow \neg 1$ holds because a model of $exp(\mathcal{HC})$ witnesses domain elements that every semantic solution of $\mathcal{HC}$ has to contain, but which violate at least one clause of the form $C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow false$, implying that no semantic solution can exist. $3 \Rightarrow 2$ is shown by encoding $\mathcal{HC}$ into a disjunctive interpolation problem (Sect. 4), which can solved with the help of Theorem 1. To this end, clauses are first duplicated to obtain a problem that is body disjoint, and subsequently normalised as described in Sect. 4.1. More details are given in Appendix A of [25]. □

5.1 The Complexity of Recursion-Free Horn Clauses

Theorem 2 gives rise to a general algorithm for (syntactically) solving recursion-free sets $\mathcal{HC}$ of Horn clauses, over constraint languages for which interpolation procedures are available. The general algorithm requires, however, to generate and solve the expansion $exp(\mathcal{HC})$ of the Horn clauses, which can be exponentially bigger than $\mathcal{HC}$ (in case $\mathcal{HC}$ is not body disjoint), and might therefore require exponential time. This leads to the question whether more efficient algorithms are possible for solving Horn clauses.

We give a number of complexity results about (semantic) Horn clause solvability; proofs of the results are given in the Appendix of [25]. Most importantly, we can observe that solvability is PSPACE-hard, for every non-trivial constraint language *Constr*. The authors of [18] conjecture a similar complexity result for the case of programs with procedures.

Lemma 1. *Suppose a constraint language can distinguish at least two values, i.e., there are two ground terms t_0 and t_1 such that $t_0 \neq t_1$ is satisfiable. Then the semantic solvability problem for recursion-free Horn clauses is PSPACE-hard.*

Looking for upper bounds, it is easy to see that solvability of Horn clauses is in co-NEXPTIME for any constraint language with satisfiability problem in NP (for instance, quantifier-free Presburger arithmetic). This is because the size of the expansion $exp(\mathcal{HC})$ is at most exponential in the size of $\mathcal{HC}$. Individual constraint languages admit more efficient solvability checks:

Theorem 3. *Semantic solvability of recursion-free Horn clauses over the constraint language of Booleans is PSPACE-complete.*

Constraint languages that are more expressive than the Booleans lead to a significant increase in the complexity of solving Horn clauses. The lower bound in the following theorem can be shown by simulating time-bounded non-deterministic Turing machines.

Theorem 4. *Semantic solvability of recursion-free Horn clauses over the constraint language of quantifier-free Presburger arithmetic is co-NEXPTIME-complete.*

The lower bounds in Lemma 1 and Theorem 4 hinge on the fact that sets of Horn clauses can contain shared relation symbols in bodies. Neither result holds if we restrict attention to body-disjoint Horn clauses, which correspond to disjunctive interpolation as introduced in Sect. 4. Since the expansion $exp(\mathcal{HC})$ of body-disjoint Horn clauses is linear in the size of the set of Horn clauses, also solvability can be checked efficiently:

Theorem 5. *Semantic solvability of a set of body-disjoint Horn clauses, and equivalently the existence of a solution for a disjunctive interpolation problem, is in co-NP when working over the constraint languages of Booleans and quantifier-free Presburger arithmetic.*

Body-disjoint Horn clauses are still expressive: they can directly encode acyclic control-flow graphs, as well as acyclic unfolding of many simple recursion patterns.

For proofs of all results of this section, please consult [25].

6 Model Checking with Recursive Horn Clauses

Whereas *recursion-free* Horn clauses generalise the concept of Craig interpolation, solving *recursive* Horn clauses corresponds to the verification of general programs with loops, recursion, or concurrency features [9]. Procedures to solve recursion-free Horn clauses can serve as a building block within model checking algorithms for recursive Horn clauses [9], and are used to construct or refine abstractions by analysing spurious counterexamples. In particular, our disjunctive interpolation can be used for this purpose, and offers a high degree of flexibility due to the possibility to analyse counterexamples combining multiple execution traces. We illustrate the use of disjunctive interpolation within a predicate abstraction-based algorithm for solving Horn clauses. Our model checking algorithm is similar in spirit to the procedure in [9], and is explained in Sect. 6.1.

And/or trees of clauses. For sake of presentation, in our algorithm we represent counterexamples (i.e., recursion-free sets of Horn clauses) in the form of and/or trees labelled with clauses. Such trees are defined by the following grammar:

$$AOTree \;::=\; And(h, AOTree, \ldots, AOTree) \;\mid\; Or(AOTree, \ldots, AOTree)$$

where h ranges over (possibly recursive) Horn clauses. We only consider well-formed trees, in which the children of every *And*-node have head symbols that are consistent with the body literals of the clause stored in the node, and the sub-trees of an *Or*-node all have the same head symbol. And/or trees are turned into body-disjoint recursion-free sets of clauses by renaming relation symbols appropriately.

Example 5. The clauses in Fig. 2 can be represented by the following and/or tree (referring to clauses in Fig. 1).

$$And\Big((4),\ Or(And((1)),\ And((2),\ And((1))),\ And((3), And((1))))\Big)$$

Solving and/or dags. Counterexamples extracted from model checking problems often assume the form of and/or *dags*, rather than and/or *trees*. Since and/or-dags correspond to Horn clauses that are not body-disjoint, the complexity-theoretic results of the last section imply that it is in general impossible to avoid the expansion of and/or-dags to and/or-trees; there are, however, various effective techniques to speed-up handling of and/or-dags (related to the techniques in [18]). We highlight two of the techniques we use in our interpolation engine Princess [4], which we used in our experimental evaluation of the next section:

1) counterexample-guided expansion expands and/or-dags lazily, until an unsatisfiable fragment of the fully expanded tree has been found; such a fragment is sufficient to compute a solution. Counterexamples are useful in two ways: they can determine which or-branch of an and/or-dag is still satisfiable and has to be expanded further, but also whether it is necessary to create further copies of a shared subtree.

2) and/or dag restructuring factors out common sub-dags underneath an *Or*-node, making the and/or-dag more tree-like.

6.1 A Predicate Abstraction-Based Model Checking Algorithm

Our model checking algorithm is in Fig. 3, and similar in spirit as the procedure in [9]; it has been implemented in the model checker Eldarica.[2] Solutions for Horn clauses are constructed in disjunctive normal form by building an abstract reachability graph over a set of given predicates. When a counterexample is detected (a clause with consistent body literals and head *false*), a theorem prover is used to verify that the counterexample is genuine; spurious counterexamples are eliminated by generating additional predicates by means of disjunctive interpolation.

In Fig. 3, $\Pi : \mathcal{R} \to \mathcal{P}_{\text{fin}}(Constr)$ denotes a mapping from relation symbols to the current (finite) set of predicates used to approximate the relation symbol. Given a (possibly recursive) set $\mathcal{HC}$ of Horn clauses, we define an *abstract reachability graph* (ARG) as a hyper-graph (S, E), where

[2] `http://lara.epfl.ch/w/eldarica`

- $S \subseteq \{(p, Q) \mid p \in \mathcal{R}, Q \subseteq \Pi(p)\}$ is the set of nodes, each of which is a pair consisting of a relation symbol and a set of predicates.
- $E \subseteq S^* \times \mathcal{HC} \times S$ is the hyper-edge relation, with each edge being labelled with a clause. An edge $E(\langle s_1, \ldots, s_n \rangle, h, s)$, with $h = (C \wedge B_1 \wedge \cdots \wedge B_n \rightarrow H) \in \mathcal{HC}$, implies that
  - $s_i = (p_i, Q_i)$ and $B_i = p_i(\bar{t}_i)$ for all $i = 1, \ldots, n$, and
  - $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$, where we write $Q_i[\bar{t}_i]$ for the conjunction of the predicates Q_i instantiated for the argument terms t_i.

An ARG (S, E) is called *closed* if the edge relation represents all Horn clauses in $\mathcal{HC}$. This means, for every clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow H) \in \mathcal{HC}$ and every sequence $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$ of nodes one of the following properties holds:

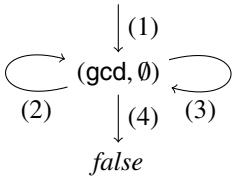
- $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \mathit{false}$, or
- there is an edge $E(\langle (p_1, Q_1), \ldots, (p_n, Q_n) \rangle, C, s)$ such that $s = (p, Q)$, $H = p(\bar{t})$, and $Q = \{\phi \in \Pi(p) \mid C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \models \phi[\bar{t}]\}$.

Lemma 2. *A set $\mathcal{HC}$ of Horn clauses has a closed ARG (S, E) if and only if $\mathcal{HC}$ is syntactically solvable.*

A proof is given in Appendix F of [25]. The function ExtractCEX extracts an and/or-tree representing a set of counterexamples, which can be turned into a recursion-free body-disjoint set of Horn clauses, and solved as described in Sect. 4.1. In general, the tree contains both conjunctions (from clauses with multiple body literals) and disjunctions, generated when following multiple hyper-edges (the case $|T| > 1$). Disjunctions make it possible to eliminate multiple counterexamples simultaneously. The algorithm is parametric in the precise strategy used to compute counterexamples (represented as non-deterministic choice in the pseudo code). The strategies we evaluated in the experiments (shown in the next section) are:

TI extraction of a single counterexamples with minimal depth (which means that disjunctive interpolation reduces to **T**ree **I**nterpolation), and
DI simultaneous extraction of all counterexamples with minimal depth (so that genuine **D**isjunctive **I**nterpolation is used).

Example 6. We consider the Horn clauses given in Fig. 1, Sect. 2. Starting with an empty predicate map Π, the function ConstructARG will construct the reachability graph shown on the right (edges are labelled with the clauses from Fig. 1). Since *false* is reachable, function ExtractCEX will be called to extract a counterexample; possible results of executing ExtractCEX include:



$$tree_1 = And((4),\ And((1))),$$
$$tree_2 = And((4),\ Or(And((1)),\ And((2), And((1))),\ And((3), And((1)))))$$

The counterexample $tree_2$ corresponds to the clauses shown in Fig. 2. Elimination of this counterexample with the help of disjunctive interpolation yields the predicates discussed in Example 4, which are sufficient to construct a closed ARG.

$S := \emptyset,\ E := \emptyset,\ \Pi := \{p \mapsto \emptyset \mid p \in \mathcal{R}\}$ ▷ Empty graph, no predicates
function ConstructARG
  while *true* **do**
    pick clause $h = (C \wedge p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \rightarrow H) \in \mathcal{HC}$
      and nodes $(p_1, Q_1), \ldots, (p_n, Q_n) \in S$
      such that $\neg\exists s.\ (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, s) \in E$
      and $C \wedge Q_1[\bar{t}_1] \wedge \cdots \wedge Q_n[\bar{t}_n] \not\models \mathit{false}$
    if no such clauses and nodes exist **then return** $\mathcal{HC}$ is solvable
    if $H = \mathit{false}$ **then** ▷ Refinement needed
      $\mathit{tree} := \mathit{And}(h, \text{ExtractCEX}(p_1, Q_1), \ldots, \text{ExtractCEX}(p_n, Q_n))$
      if *tree* is unsatisfiable **then**
        extract disjunctive interpolant from *tree*, add predicates to Π
        delete part of (S, E) used to construct *tree*
      else return $\mathcal{HC}$ is unsolvable, with counterexample trace *tree*
    else ▷ Add edge to ARG
      then $H = p(\bar{t})$
      $Q := \{\phi \in \Pi(p) \mid \{C\} \cup Q_1 \cup \ldots \cup Q_n \models \phi\}$
      $e := (\langle (p_1, Q_1), \ldots, (p_n, Q_n)\rangle, h, (p, Q))$
      $S := S \cup \{(p, Q)\},\ E := E \cup \{e\}$

function ExtractCEX($\mathit{root} : S$) ▷ Extract disjunctive interpolation problem
  pick $\emptyset \neq T \subseteq E$ with $\forall e \in T.\ e = (\_, \_, \mathit{root})$
  return $\mathit{Or}\{\mathit{And}(h, \text{ExtractCEX}(s_1), \ldots, \text{ExtractCEX}(s_n)) \mid (\langle s_1, \ldots, s_n\rangle, h, \mathit{root}) \in T\}$

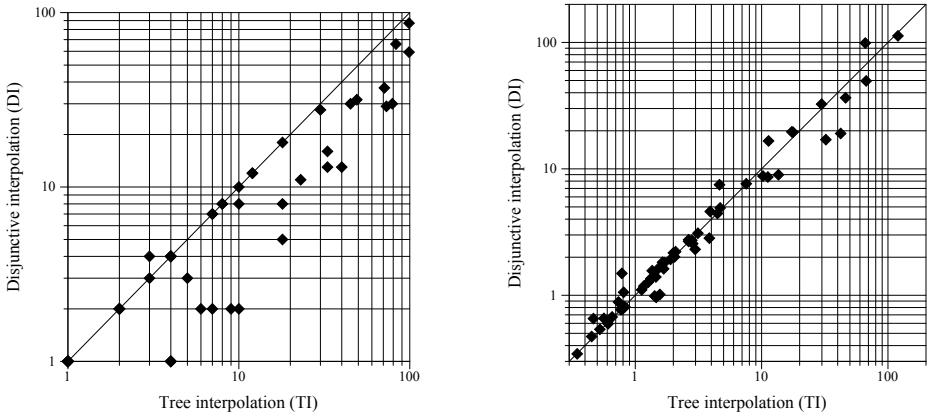
Fig. 3. Algorithm for construction of abstract reachability graphs

We remark that we have also implemented a simpler "global" algorithm that approximates each relation symbol globally with a single conjunction of inferred predicates instead of disjunction of conjunctions. The two algorithms behave similarly in our experience, with the global one occasionally slower, but conceptually simpler. What allowed us to use a simpler algorithm is precisely the more general form of the interpolation. This shows another advantage of more expressive interpolation: the simplicity of verification algorithms we can build on top of it.

6.2 Experimental Evaluation

We have evaluated our algorithm on a set of benchmarks in integer linear arithmetic from the NTS library [16] translated into Horn clauses [3]. These include recursive algorithms, benchmarks extracted from programs with singly-linked lists, VHDL models of circuits, verification conditions for programs with arrays, benchmarks from the NECLA static analysis suite, and C programs with asynchronous procedure calls translated using the approach of [7]. Scatter plots comparing the results for the **T**ree **I**nterpolation and **D**isjunctive **I**nterpolation runs are given in Fig. 4. A table with detailed data is provided in [25]. The experiments show comparable verification times and performance for tree interpolation and disjunctive interpolation runs. Studying the results more closely, we observed that **DI** consistently led to a smaller number of abstraction refinement steps

[3] `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/clauses/LIA/`



(a) Number of CEGAR iterations (b) Model checking runtime

Fig. 4. Comparison of the number of required refinement steps, and the runtime (in seconds), for the case of single counterexamples (**TI**) and simultaneous extraction of all minimal-depth counterexamples (**DI**). All experiments were done on an Intel Core i5 2-core machine with 3.2GHz and 8Gb, with a timeout of 900s.

(the scatter plot in Fig. 4); this indicates that **DI** is indeed able to eliminate multiple counterexamples simultaneously, and to rapidly generate predicates that are useful for abstraction. The experiments also showed that there is a trade-off between the time spent generating predicates, and the quality of the predicates. In **TI**, on average 31% of the verification is used for predicate generation (interpolation), while with **DI** 42% is used; in some of the benchmarks from [7], this led to the phenomenon that **DI** was slower than **TI**, despite fewer refinement steps. This may change as we make further improvements to our prototype implementation of disjunctive interpolation. We also compared our results to the performance of HSF,[4] a state-of-the-art verification engine for Horn clauses. HSF was faster on average, but for harder examples [7] our tool was comparable (see the technical report for detailed results).

Conclusions

We have introduced disjunctive interpolation as a new form of Craig interpolation tailored to model checkers based on Horn clauses. Disjunctive interpolation can be identified as solving body-disjoint systems of recursion-free Horn clauses, and subsumes a number of previous forms of interpolation, including tree interpolation. We believe that the flexibility of disjunctive interpolation is highly beneficial for building interpolation-based model checkers. We expect further performance improvements from better implementation of disjunctive interpolation and better techniques to select sets of counterexample paths given to interpolation.

[4] `http://www7.in.tum.de/tools/hsf/`

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)
2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: WHALE: An interpolation-based algorithm for inter-procedural verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012)
3. Ball, T., Podelski, A., Rajamani, S.K.: Relative completeness of abstraction refinement for software model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 158–172. Springer, Heidelberg (2002)
4. Brillout, A., Kroening, D., Rümmer, P., Wahl, T.: An interpolating sequent calculus for quantifier-free Presburger arithmetic. Journal of Automated Reasoning 47, 341–367 (2011)
5. Cimatti, A., Griggio, A., Sebastiani, R.: Efficient generation of Craig interpolants in satisfiability modulo theories. ACM Trans. Comput. Log. 12(1), 7 (2010)
6. Craig, W.: Linear reasoning. A new form of the Herbrand-Gentzen theorem. The Journal of Symbolic Logic 22(3), 250–268 (1957)
7. Ganty, P., Majumdar, R.: Algorithmic verification of asynchronous programs. CoRR, abs/1011.0551 (2010)
8. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
9. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
10. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. In: POPL (2011)
11. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 188–203. Springer, Heidelberg (2011)
12. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
13. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244. ACM (2004)
14. Hoder, K., Bjørner, N.: Generalized property directed reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
15. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 187–202. Springer, Heidelberg (2012)
16. Hojjat, H., Konečný, F., Garnier, F., Iosif, R., Kuncak, V., Rümmer, P.: A verification toolkit for numerical transition systems (tool paper). In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 247–251. Springer, Heidelberg (2012)
17. Jhala, R., Majumdar, R., Rybalchenko, A.: HMC: Verifying functional programs using abstract interpreters. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 470–485. Springer, Heidelberg (2011)
18. Lal, A., Qadeer, S., Lahiri, S.K.: A solver for reachability modulo theories. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 427–443. Springer, Heidelberg (2012)
19. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
20. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
21. McMillan, K.L., Rybalchenko, A.: Solving constrained Horn clauses using interpolation. Technical Report MSR-TR-2013-6, Microsoft Research (January 2013)

22. Méndez-Lojo, M., Navas, J., Hermenegildo, M.V.: A flexible (C)LP-based approach to the analysis of object-oriented programs. In: King, A. (ed.) LOPSTR 2007. LNCS, vol. 4915, pp. 154–168. Springer, Heidelberg (2008)
23. Peralta, J.C., Gallagher, J.P., Saglam, H.: Analysis of imperative programs through analysis of constraint logic programs. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 246–261. Springer, Heidelberg (1998)
24. Rümmer, P., Hojjat, H., Kuncak, V.: Classifying and solving horn clauses for verification. In: VSTTE (2013)
25. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification (extended technical report). CoRR, abs/1301.4973 (2013)
26. Sery, O., Fedyukovich, G., Sharygina, N.: Interpolation-based function summaries in bounded model checking. In: Eder, K., Lourenço, J., Shehory, O. (eds.) HVC 2011. LNCS, vol. 7261, pp. 160–175. Springer, Heidelberg (2012)
27. Terauchi, T.: Dependent types from counterexamples. In: Hermenegildo, M.V., Palsberg, J. (eds.) POPL, pp. 119–130. ACM (2010)
28. Unno, H., Terauchi, T., Kobayashi, N.: Automating relatively complete verification of higher-order functional programs. In: POPL (2013)

Generating Non-linear Interpolants by Semidefinite Programming*

Liyun Dai[1,3], Bican Xia[1], and Naijun Zhan[2]

[1] LMAM & School of Mathematical Sciences, Peking University
[2] State Key Laboratory of Computer Science, Institute of Software, CAS
[3] Beijing International Center for Mathematical Research, Peking University
dailiyun@pku.edu.cn, xbc@math.pku.edu.cn, znj@ios.ac.cn

Abstract. Interpolation-based techniques have been widely and successfully applied in the verification of hardware and software, e.g., in bounded-model checking, CEGAR, SMT, etc., in which the hardest part is how to synthesize interpolants. Various work for discovering interpolants for propositional logic, quantifier-free fragments of first-order theories and their combinations have been proposed. However, little work focuses on discovering polynomial interpolants in the literature. In this paper, we provide an approach for constructing non-linear interpolants based on semidefinite programming, and show how to apply such results to the verification of programs by examples.

Keywords: Craig interpolant, Positivstellensatz Theorem, semidefinite programming, program verification.

1 Introduction

It becomes a grand challenge to guarantee the correctness of software, as our modern life depends more and more on computerized systems. There are lots of verification techniques based either on model-checking [1], theorem proving [2,3], abstract interpretation [4] or their combination, which have been invented for the verification of hardware and software, like bounded model-checking [5], CEGAR [6], satisfiability modulo theories (SMT) [7], etc. Scalability is a bottleneck of these techniques, as many of real softwares are very complex with different features like complicated data structures, concurrency, distributed, real-time and hybrid etc. Interpolation-based techniques provide a powerful mechanism for local and modular reasoning, which indeed improves the scalability of these techniques, in which the notion of Craig interpolants plays a key role.

Interpolation-based local and modular reasoning was first applied in theorem proving by Nelson and Oppen [8], called Nelson-Oppen method. The basic idea of Nelson-Oppen method is to reduce the satisfiability (validity) of a composite theory into the ones of its component theories whose satisfiability (validity) have been obtained. The hardest part of the method, which also determines the efficiency of the method, is to construct a formula using the common part of the component theories for a given formula

* The first two authors are funded by NSFC-11271034, NSFC-11290141 and SYSKF1207, and the third author is funded by NSFC-91118007, NSFC-60970031 and 2012ZX03039-004.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 364–380, 2013.

of the composite theory with Craig's Interpolation Theorem [9]. In the past decade, the Nelson-Oppen method was further extended to SMT which is based on DPLL [10] and Craig's Interpolation Theorem for combining different decision procedures in order to verify a property of programs with complicated data structures. For instance, Z3 [11] integrates more than 10 different decision procedures up to now, including propositional logic, equality logic with uninterpreted functions, Presburger arithmetic, array logic, difference arithmetic, bit vector logic etc.

In recent years, it is noted that interpolation based local and modular reasoning is quite helpful to improve the scalability of model-checking, in particular for bounded model-checking of systems with finite or infinite states [5,12,13], CEGAR [14,15], etc. McMillan first considered how to combine Craig interpolants with bounded model-checking to verify infinite state systems [12]. The basic idea of his approach is to generate invariants using Craig interpolants, so that it can be claimed that an infinite state system satisfies a property after k steps in model-checking whenever an invariant, which is strong enough to guarantee the property, is obtained. In [14,15,16], how to apply the local property of Craig interpolants generated from a counter-example to refine the abstract model in order to exclude the spurious counter-example in CEGAR was investigated. Meanwhile, in [17], using interpolation technique to generate a set of atomic predicates as the base of machine-learning based verification technique was investigated by Wang et al.

Obviously, synthesizing Craig interpolants is the cornerstone of interpolation based techniques. In fact, many approaches have been proposed in the literature. In [13], McMillan presented a method for deriving Craig interpolants from proofs in the quantifier-free theory of linear inequality and uninterpreted function symbols, and based on which an interpolating theorem prover was provided. In [15], Henzinger et al. proposed a method to synthesize Craig interpolants for a theory with arithmetic and pointer expressions, as well as call-by-value functions. In [18], Yorsh and Musuvathi presented a combination method to generate Craig interpolants for a class of first-order theories. In [19], Kapur et al presented different efficient procedures to construct interpolants for the theories of arrays, sets and multisets using the reduction approach. Rybalchenko and Sofronie-Stokkermans [20] proposed an approach to reducing the synthesis of Craig interpolants of the combined theory of linear arithmetic and uninterpreted function symbols to constraint solving.

However, in the literature, there is little work on how to synthesize non-linear interpolants, except that in [21] Kupferschmid and Becker provided a method to construct non-linear Craig Interpolant using iSAT, which is a variant of SMT solver based on interval arithmetic.

In this paper we investigate how to construct non-linear interpolants. The idea of our approach is as follows: Firstly, we reduce the problem of generating interpolants for two arbitrary polynomial formulas to that of generating interpolants for two semi-algebraic systems (SASs), which is a conjunction of a set of polynomial equations, inequations and inequalities (see the definition later). Then, by **Positivstellensatz Theorem** [22], there exists a witness to indicate that the considered two SASs do not have common real solutions if their conjunction is unsatisfiable. Parrilo in [23,24] gave an approach for constructing the witness by applying semidefinite programming [25]. Our algorithm

invokes Parrilo's method as a subroutine. Our purpose is to construct Craig interpolants, so we need to obtain a special witness. In general, we cannot guarantee the existence of the special witness, which means that our approach is sound but incomplete. However, we discuss that if the considered two SASs meet the *Archimedean condition*, (e.g. each variable occurring in the SASs is bounded, which is a reasonable assumption in practice), our approach is not only sound, but also complete. We demonstrate our approach by some examples, in particular, we show how to apply the results to program verification by examples.

The complexity of our approach is polynomial in $ud\binom{n+d/2}{n}\binom{n+d}{n}$, where u is the number of polynomial constraints in the considered problem, n is the number of variables, and d is the highest degree of polynomials and interpolants. So, the complexity of our approach is polynomial in d for a given problem in which n and u are fixed.

Structure of the Paper: The rest of the paper is organized as follows. By a running example, we sketch our approach and show how to apply it to program verification in Section 2. Some necessary preliminaries are introduced in Section 3. A sound but incomplete algorithm for synthesizing non-linear interpolants in general case is described in Section 4. Section 5 provides a practical algorithm for systems containing only non-strict inequalities and satisfying the Archimedean condition. Section 6 focuses on the correctness and complexity analysis of our approach. Our implementation and experimental results are briefly reported in Section 7. Section 8 summarizes the paper and discusses future work.

2 An Overview of Our Approach

In this section, we sketch our approach and show how to apply our results to program verification by an example.

1	IF $(x*x+y*y<1)$	$g_1 = 1-x^2-y^2>0$
2	{ /* initial values */	
3	WHILE $(x*x+y*y<3)$	$g_2 = 3-x^2-y^2>0$
4	{ $x := x*x+y-1;$	$f_1 = x^2+y-1-x'=0$
5	$y := y+x*y+1;$	$f_2 = y+x'y+1-y'=0$
6	IF $(x*x-2*y*y-4>0)$	$g_3 = x'^2-2y'^2-4>0$
7	/* unsafe area */	
8	**error**(); } }	

Code 1.1

Consider the program in Code 1.1 (left part). This program tests the initial value of x and y at line 1, afterwards executes the *while loop* with $x^2+y^2<3$ as the loop condition. The body of the while loop contains two assignments and an **if** statement in sequence. The property we wish to check is that **error()** procedure will never be executed. Suppose there is an execution $1\to 3\to 4\to 5\to 6\to 8$. We can encode such an execution by the formulas in Code 1.1 (right part). Note that in these formulas we use unprimed and primed versions of each variable to represent the values of the variable before and after

updating respectively. Obviously, the execution is infeasible iff the conjunction of these formulas is unsatisfiable. Let $\phi \triangleq g_1 > 0 \wedge f_1 = 0 \wedge f_2 = 0$[1] and $\psi \triangleq g_3 > 0$. To show $\phi \wedge \psi$ is unsatisfiable, we need to construct an *interpolant* θ for ϕ and ψ, i.e., $\phi \Rightarrow \theta$ and $\theta \Rightarrow \neg\psi$. If there exist $\delta_1, \delta_2, \delta_3, h_1, h_2$ such that

$$g_1\delta_1 + f_1h_1 + f_2h_2 + g_3\delta_2 + \delta_3 = -1,$$

where $\delta_1, \delta_2, \delta_3 \in \mathbb{R}[x, y, x', y']$ are sums of squares and $h_1, h_2 \in \mathbb{R}[x, y, x', y']$, then $\theta \triangleq g_3\delta_2 + \frac{1}{2} \leq 0$ is such an interpolant for ϕ and ψ. In this example, applying our tool `AiSat`, we obtain in 0.025 seconds that

$$\begin{aligned}
h_1 &= -290.17 - 56.86y' + 1109.95x' + 37.59y - 32.20yy' + 386.77yx' + 203.88y^2 + 107.91x^2,\\
h_2 &= -65.71 + 0.39y' + 244.14x' + 274.80y + 69.33yy' - 193.42yx' - 88.18y^2 - 105.63x^2,\\
\delta_1 &= 797.74 - 31.38y' + 466.12y'^2 + 506.26x' + 79.87x'y' + 402.44x'^2 + 104.43y\\
&\quad +41.09yy' - 70.14yx' + 451.64y^2 + 578.94x^2\\
\delta_2 &= 436.45,\\
\delta_3 &= 722.62 - 91.59y' + 407.17y'^2 + 69.39x' + 107.41x'y' + 271.06x'^2 + 14.23y + 188.65yy'\\
&\quad +69.33yy'^2 - 600.47yx' - 226.01yx'y' + 142.62yx'^2 + 325.78y^2 - 156.69y^2y' + 466.12y^2y'^2\\
&\quad +10.54y^2x'y' + 595.87y^2x'^2 - 11.26y^3 + 41.09y^3y' + 18.04y^3x' + 451.64y^4 + 722.52x^2\\
&\quad -80.15x^2y' + 466.12x^2y'^2 - 495.78x^2x' + 79.87x^2x'y' + 402.44x^2x'^2 + 64.57x^2y\\
&\quad +241.99y^2x' + 73.29x^2yy' - 351.27x^2yx' + 826.70x^2y^2 + 471.03x^4.
\end{aligned}$$

Note that δ_1 can be represented as $923.42(0.90 + 0.7y - 0.1y' + 0.43x')^2 + 252.84(0.42 - 0.28y + 0.21y' - 0.84x')^2 + 461.69(-0.1 - 0.83y + 0.44y' + 0.34x')^2 + 478(-0.06 + 0.48y + 0.87y' + 0.03x')^2 + 578.94(x)^2$. Similarly, δ_2 and δ_3 can be represented as sums of squares also.

Moreover, using the approach in [26], we can prove θ is an inductive invariant of the loop, therefore, **error()** will never be executed.

3 Theoretical Foundations

In this section, for self-containedness, we briefly introduce some basic notions and mathematical theories, based on which our approach is developed.

3.1 Problem Description

Definition 1 (Interpolants). *A theory $\mathcal{T}$ has interpolant if for all formulae ϕ and ψ in the signature of $\mathcal{T}$, if $\models_{\mathcal{T}} (\phi \wedge \psi) \Rightarrow \bot$, then there exists a formula Θ that contains only symbols that ϕ and ψ share such that $\phi \models_{\mathcal{T}} \Theta$ and $\models_{\mathcal{T}} (\psi \wedge \Theta) \Rightarrow \bot$.*

In what follows, we denote by $\mathbf{x}$ a variable vector[2] $(x_1, \cdots, x_n)$ in $\mathbb{R}^n$, and by $\mathbb{R}[\mathbf{x}]$ the polynomial ring with real coefficients in variables $\mathbf{x}$.

A semi-algebraic system (SAS) $\mathcal{T}(\mathbf{x})$ is of the form $\bigwedge_{j=0}^{k} f_j(\mathbf{x}) \triangleright_j 0$, where f_j are polynomials in $\mathbb{R}[\mathbf{x}]$ and $\triangleright_j \in \{=, \neq, \geq\}$. Clearly, any polynomial formula ϕ can be represented as the disjunction of several SASs. Let $\phi_1 = \bigvee_{t=1}^{m} \mathcal{T}_{1t}(\mathbf{x}), \phi_2 = \bigvee_{l=1}^{n} \mathcal{T}_{2l}(\mathbf{x})$ be

[1] As $g_1 > 0 \Rightarrow g_2 > 0$, we ignore $g_2 > 0$ in ϕ.

[2] In the following, we also abuse $\mathbf{x}$ to denote the set of variables $\{x_1, \ldots, x_n\}$.

two polynomial formulas in $\mathbb{R}[\mathbf{x}]$, and $\phi_1 \wedge \phi_2 \models \bot$, i.e., ϕ_1 and ϕ_2 do not share any real solutions. Then, the problem to be considered in this paper is how to find another polynomial formula I such that $\phi_1 \models I$ and $I \wedge \phi_2 \models \bot$.

It is easy to show that if, for each t and l, there is an interpolant I_{tl} for SASs $\mathcal{T}_{1t}(\mathbf{x})$ and $\mathcal{T}_{2l}(\mathbf{x})$, then $I = \bigvee_{t=1}^{m} \bigwedge_{l=1}^{n} I_{tl}$ is an interpolant of ϕ_1 and ϕ_2. Thus, we only need to consider how to construct interpolants for two SASs in the rest of this paper.

Discussions on Common Variables: In reality, it is more likely that ϕ_1 and ϕ_2 in the above problem description are over different sets of variables, say over $\mathbf{x}_1$ and $\mathbf{x}_2$ respectively, and $\mathbf{x}_1 \neq \mathbf{x}_2$. So, we need to reduce the interpolant generation of ϕ_1 and ϕ_2 to that of another two formulas that share the common variables first. A simple way to achieve this is through introducing $\exists \mathbf{x}_1 - \mathbf{x}_2$ over $\phi_1(\mathbf{x}_1)$ and $\exists \mathbf{x}_2 - \mathbf{x}_1$ over $\phi_2(\mathbf{x}_2)$, then apply quantifier elimination to $\exists \mathbf{x}_1 - \mathbf{x}_2.\phi_1(\mathbf{x}_1)$ and $\exists \mathbf{x}_2 - \mathbf{x}_1.\phi_2(\mathbf{x}_2)$, and obtain two formulas on the common variables $\mathbf{x}_1 \cap \mathbf{x}_2$. Obviously, the interpolant generation problem of ϕ_1 and ϕ_2 is reduced to that of the two resulted formulas.

But in practice, as the cost of quantifier elimination is very high, we can adopt the following more efficient way. For each $x \in \mathbf{x}_1 - \mathbf{x}_2$, if x is a local variable introduced in the respective program, we always have an equation $x = h$ corresponding to the assignment to x (possibly the composition of a sequence of assignments to x); otherwise, x is a global variable, but only occurring in ϕ_1. For this case, we introduce an equation $x = x$ to ϕ_2; Symmetrically, each $x \in \mathbf{x}_2 - \mathbf{x}_1$ can be coped with similarly. The detailed discussion can be found in the full version of this paper [27].

So, in what follows, we assume any two SASs share the common variables if no otherwise stated.

3.2 Real Algebraic Geometry

In this subsection, we introduce some basic notions and results on real algebraic geometry, that will be used later.

Definition 2 (ideal). *Let $\mathcal{I}$ be an ideal in $\mathbb{R}[\mathbf{x}]$, that is, $\mathcal{I}$ is an additive subgroup of $\mathbb{R}[\mathbf{x}]$ satisfying $fg \in \mathcal{I}$ whenever $f \in \mathcal{I}$ and $g \in \mathbb{R}[\mathbf{x}]$. Given $h_1, \ldots, h_m \in \mathbb{R}[\mathbf{x}]$, $\langle h_1, \ldots, h_m \rangle = \left\{ \sum_{j=1}^{m} u_j h_j \mid u_1, \ldots, u_m \in \mathbb{R}[\mathbf{x}] \right\}$ denotes the ideal generated by $h_1, \ldots, h_m$.*

Definition 3 (multiplicative monoid). *Given a polynomial set P, let $Mult(P)$ be the* multiplicative monoid *generated by P, i.e., the set of finite products of the elements of P (including the empty product which is defined to be 1).*

Definition 4 (Cone). *A* cone *$\mathcal{C}$ of $\mathbb{R}[\mathbf{x}]$ is a subset of $\mathbb{R}[\mathbf{x}]$ satisfying the following conditions: **(i)** $p_1, p_2 \in \mathcal{C} \Rightarrow p_1 + p_2 \in \mathcal{C}$; **(ii)** $p_1, p_2 \in \mathcal{C} \Rightarrow p_1 p_2 \in \mathcal{C}$; **(iii)** $p \in \mathbb{R}[\mathbf{x}] \Rightarrow p^2 \in \mathcal{C}$.*

Given a set $P \subseteq \mathbb{R}[\mathbf{x}]$, let $\mathcal{C}(P)$ be the smallest cone of $\mathbb{R}[\mathbf{x}]$ that contains P. It is easy to see that $\mathcal{C}(\emptyset)$ corresponds to the polynomials that can be represented as a sum of squares, and is the smallest cone in $\mathbb{R}[\mathbf{x}]$, i.e., $\left\{ \sum_{i=1}^{s} p_i^2 \mid p_1, \ldots, p_s \in \mathbb{R}[\mathbf{x}] \right\}$, denoted by $\mathbf{SOS}$. For a finite set $P \subseteq \mathbb{R}[\mathbf{x}]$, $\mathcal{C}(P)$ can be represented as:

$$\mathcal{C}(P) = \{ \sum_{i=1}^{r} q_i p_i \mid q_1, \ldots, q_r \in \mathcal{C}(\emptyset), p_1, \ldots, p_r \in Mult(P) \}.$$

Positivstellensatz Theorem, due to Stengle [22], is an important theorem in real algebraic geometry. It states that, for a given SAS, either the system has real solution(s), or there exists a polynomial to indicate that the system has no solution.

Theorem 1 (Positivestellensatz Theorem, [22]). *Let* $(f_j)_{j=1}^s$, $(g_k)_{k=1}^t, (h_l)_{l=1}^u$ *be finite families of polynomials in* $\mathbb{R}[\mathbf{x}]$. *Denote by* $\mathcal{C}$ *the cone generated by* $(f_j)_{j=1}^s$, *Mult the multiplicative monoid generated by* $(g_k)_{k=1}^t$, *and* $\mathcal{I}$ *the ideal generated by* $(h_l)_{l=1}^u$. *Then the following two statements are equivalent:*

1. *the SAS* $\begin{cases} f_1(\mathbf{x}) \geq 0, & \cdots, \quad f_s(\mathbf{x}) \geq 0, \\ g_1(\mathbf{x}) \neq 0, & \cdots, \quad g_t(\mathbf{x}) \neq 0, \\ h_1(\mathbf{x}) = 0, & \cdots, \quad h_u(\mathbf{x}) = 0 \end{cases}$ *has no real solutions;*
2. *there exist* $f \in \mathcal{C}$, $g \in$ *Mult*, $h \in \mathcal{I}$ *such that* $f + g^2 + h \equiv 0$.

3.3 Semidefinite Programming

In [22], Stengle did not provide a constructive proof for Theorem 1. However, Parrilo in [23,24] provided a constructive way to obtain the witness, which is based on semidefinite programming. Parrilo's result will be the starting point of our method, so we briefly review semidefinite programming below. We use Sym_n to denote the set of $n \times n$ real symmetric matrices, and $\deg(f)$ the highest total degree of f for a given polynomial f in the sequel.

Definition 5 (Positive semidefinite matrix). *A matrix* $M \in Sym_n$ *is called* positive semidefinite*, denoted by* $M \succeq 0$, *if* $\mathbf{x}^T M \mathbf{x} \geq 0$ *for all* $\mathbf{x} \in \mathbb{R}^n$.

Definition 6. *The* inner product *of two matrices* $A = (a_{ij}), B = (b_{ij}) \in \mathbb{R}^{n \times n}$, *denoted by* $\langle A, B \rangle$, *is defined by* $Tr(A^T B) = \sum_{i,j=1}^n a_{ij} b_{ij}$.

Definition 7 (Semidefinite programming (SDP)). *The standard (primal) and dual forms of a* **SDP** *are respectively given in the following:*

$$p^* = \inf_{X \in Sym_n} \langle C, X \rangle \ \textit{s.t.}\ X \succeq 0,\ \langle A_j, X \rangle = b_j \ (j = 1, \ldots, m) \tag{1}$$

$$d^* = \sup_{\mathbf{y} \in \mathbb{R}^m} \mathbf{b}^T \mathbf{y} \ \textit{s.t.}\ \sum_{j=1}^m y_j A_j + S = C,\ S \succeq 0, \tag{2}$$

where $C, A_1, \ldots, A_m, S \in Sym_n$ *and* $\mathbf{b} \in \mathbb{R}^m$.

There are many efficient algorithms to solve **SDP** such as interior-point method. We present a basic path-following algorithm to solve (1) in the following.

Definition 8 (Interior point for SDP).

$$\begin{aligned} \textit{intF}_p &= \{X : \langle A_i, X \rangle = b_i \ (i = 1, \ldots, m),\ X \succ 0\}, \\ \textit{intF}_d &= \left\{ (\mathbf{y}, S) : S = C - \sum_{i=1}^m A_i y_i \succ 0 \right\}, \\ \textit{intF} &= \textit{intF}_p \times \textit{intF}_d. \end{aligned}$$

Obviously, $\langle C, X \rangle - \mathbf{b}^T \mathbf{y} = \langle X, S \rangle > 0$ for all $(X, \mathbf{y}, S) \in$ *intF*. Especially, we have $d^* \leq p^*$. So the soul of interior-point method to compute p^* is to reduce $\langle X, S \rangle$ incessantly and meanwhile guarantee $(X, \mathbf{y}, S) \in$ *intF*.

Algorithm 1. `Interior_Point_Method`

input : C, A_j, b_j $(j = 1, \ldots, m)$ as in (1) and a threshold c
output: p^*

Given a $(X, \mathbf{y}, S) \in$ *intF* and $XS = \mu I$;
`/* μ is a positive constant and I is the identity matrix. */`
while $\mu > c$ **do**
  $\mu = \gamma\mu$;
  `/* γ is a fixed positive constant less than one */`
  use **Newton iteration** to solve $(X, \mathbf{y}, S) \in$ *intF* with $XS = \mu I$;
end

3.4 Constructive Proof of Theorem 1 Using SDP

Given a polynomial $f(\mathbf{x})$ of degree no more than $2d$, f can be rewritten as $f = Z^T Q Z$ where Z is a vector consists of all monomials of degrees no more than d, e.g., $Z = \left[1, x_1, x_2, \ldots, x_n, x_1x_2, x_2x_3, \ldots, x_n^d\right]^T$, and $Q = \begin{pmatrix} a_1 & \frac{a_{x_1}}{2} & \cdots & \frac{a_{x_n}}{2} \\ \frac{a_{x_1}}{2} & a_{x_1^2} & \cdots & \frac{a_{x_1x_n}}{2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{a_{x_n}}{2} & \frac{a_{x_1x_n}}{2} & \cdots & a_{x_n^d} \end{pmatrix}$ is a symmetric matrix. Note that here Q is not unique in general. Moreover, $f \in \mathcal{C}(\emptyset)$ iff there is a positive semidefinite constant matrix Q such that $f(\mathbf{x}) = Z^T Q Z$. The following lemma is an obvious fact on how to use the above notations to express the polynomial multiplication.

Lemma 1. *Given polynomials $f_1, \ldots, f_n, g_1, \ldots, g_n$, assume $\sum_{i=1}^n f_i g_i = \sum_{i=1}^s c_i m_i$, where $c_i \in \mathbb{R}$ and m_is are monomials, $g_i = Z^T Q_{2i} Z$, and $Q_2 = diag(Q_{21}, \ldots, Q_{2n})$. Then there exist symmetric matrices $Q_{11}, \ldots, Q_{1s}$ such that $c_i = \langle Q_{1i}, Q_2 \rangle$, i.e., $\sum_{i=1}^n f_i g_i = \sum_{i=1}^s \langle Q_{1i}, Q_2 \rangle m_i$, in which Q_{1i} can be constructed from the coefficients of $f_1, \ldots, f_n$.*

Example 1. Let $f = a_{20}x_1^2 + a_{11}x_1x_2 + a_{02}x_2^2$ and $g = b_{00} + b_{10}x_1 + b_{01}x_2$. Then, $fg = \langle Q_{11}, Q_2 \rangle x_1^2 + \langle Q_{12}, Q_2 \rangle x_1x_2 + \langle Q_{13}, Q_2 \rangle x_2^2 + \langle Q_{14}, Q_2 \rangle x_1x_2^2 \langle Q_{15}, Q_2 \rangle x_1^2x_2 + \langle Q_{16}, Q_2 \rangle x_2^3 + \langle Q_{17}, Q_2 \rangle x_1^3$, where

$$Q_2 = \begin{pmatrix} b_{00} & \frac{b_{10}}{2} & \frac{b_{01}}{2} \\ \frac{b_{10}}{2} & 0 & 0 \\ \frac{b_{01}}{2} & 0 & 0 \end{pmatrix}, \quad Q_{11} = \begin{pmatrix} a_{20} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad Q_{12} = \begin{pmatrix} a_{11} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}, \quad Q_{13} = \begin{pmatrix} a_{02} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix},$$

$$Q_{14} = \begin{pmatrix} 0 & \frac{a_{02}}{2} & \frac{a_{11}}{2} \\ \frac{a_{02}}{2} & 0 & 0 \\ \frac{a_{11}}{2} & 0 & 0 \end{pmatrix}, Q_{15} = \begin{pmatrix} 0 & \frac{a_{11}}{2} & \frac{a_{20}}{2} \\ \frac{a_{11}}{2} & 0 & 0 \\ \frac{a_{20}}{2} & 0 & 0 \end{pmatrix}, Q_{16} = \begin{pmatrix} 0 & 0 & \frac{a_{02}}{2} \\ 0 & 0 & 0 \\ \frac{a_{02}}{2} & 0 & 0 \end{pmatrix}, Q_{17} = \begin{pmatrix} 0 & \frac{a_{02}}{2} & 0 \\ \frac{a_{02}}{2} & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}.$$

Back to Theorem 1. We show how to find $f \in \mathcal{C}$, $g \in$ *Mult*, $h \in \mathcal{I}$ such that $f + g^2 + h \equiv 0$ via **SDP** solving. First, since $f \in \mathcal{C}$, f can be written as a sum of the products of some known polynomials and some unknown SOSs. Second, $h \in \mathcal{I}(\{h_1, \ldots, h_u\})$ is equivalent to $h = h_1p_1 + \cdots + h_up_u$, which is further equivalent to $h = h_1(q_{11} - q_{12}) + \cdots + h_u(q_{u1} - q_{u2})$, where $p_i, q_{ij} \in \mathbb{R}[\mathbf{x}]$ and $q_{ij} \in \mathbf{SOS}$[3]. Third, fix an integer $d > 0$, let

[3] For example, let $q_{i1} = (\frac{1}{4}p_i + 1)^2, q_{i2} = (\frac{1}{4}p_i - 1)^2$.

Algorithm 2. `Certificate_Generation`

input : $\{f_1, \ldots, f_n\}, g, \{h_1, \ldots, h_u\}, b$
output: either $\{p_0, \ldots, p_n\}$ and $\{q_1, \ldots, q_u\}$ such that
$1 + p_0 + p_1 f_1 + \cdots + p_n f_n + g + q_1 h_1 + \cdots + q_u h_u \equiv 0$, or NULL

Let $q_{11}, q_{12}, q_{21}, q_{22}, \ldots, q_{u1}, q_{u2} \in \mathbf{SOS}$ with $\deg(q_{i1}) \leq b$ and $\deg(q_{i2}) \leq b$ be undetermined **SOS** polynomials;
Let $p_0, p_1, \ldots, p_n \in \mathbf{SOS}$ with $\deg(p_i) \leq b$ be undetermined **SOS** polynomials;
Let $f = 1 + p_0 + p_1 f_1 + \cdots + p_n f_n + g + (q_{11} - q_{12})h_1 + \cdots + (q_{u1} - q_{u2})h_u$;
for *every monomial* $m \in f$ **do**
    Let $\langle Q_m, Q \rangle = \texttt{coeff}(f, m)$;
    /* Applying Lemma 1 */
    /* `coeff`(f, m) the coefficient of monomial m in f */
    /* Z is a monomial vector that contains all monomials with coefficient 1 and degree no more than $b/2$ */
    /* $p_0 = Z^T Q_0 Z, p_1 = Z^T Q_1 Z, \ldots, p_n = Z^T Q_n Z$ */
    /* $q_{i1} = Z^T Q_{i1} Z, q_{i2} = Z^T Q_{i2} Z, i = 1, \ldots, u$ */
    /* $Q = diag(1, Q_0, Q_1, \ldots, Q_n, 1, Q_{11}, Q_{12}, \ldots, Q_{u1}, Q_{u2})$ */
end
Applying **SDP** software `CSDP` to solve whether there exists a semi-definite symmetric matrix Q *s.t.* $\langle Q_m, Q \rangle = 0$ for every monomial $m \in f$
if *the return of* `CSDP` *is feasible* **then**
    /* $q_i = q_{i1} - q_{i2}$ */
    return $\{p_0, \ldots, p_n\}, \{q_1, \ldots, q_u\}$
else
    return NULL
end

$g = (\Pi_{i=1}^{t} g_i)^d$, and then $f + g^2 + h$ can be written as $\sum_{i=1}^{l} f_i' \delta_i$, where l is a constant integer, $f_i' \in \mathbb{R}[\mathbf{x}]$ are known polynomials and $\delta_i \in \mathbf{SOS}$ are undermined **SOS** polynomials. Therefore, Theorem 1 is reduced to fixing a sufficiently large integer d and finding undetermined **SOS** polynomials δ_i occurring in f, h with degrees no more than $\deg(g^2)$, satisfying $f + g^2 + h \equiv 0$. Based on Lemma 1, this is a **SDP** problem of form (1). The constraints of the **SDP** are of the form $\langle A_j, X \rangle = 0$, where A_j and X correspond to Q_{1j} and Q_2 in Lemma 1, respectively. And Q_2 is a block diag matrix whose blocks correspond to the undetermined **SOS** polynomials in the above discussion.

Theorem 2 ([23]). *Consider a system of polynomial equalities and inequalities of the form in Theorem 1. Then the search for bounded degree Positivstellensatz refutations can be done using semidefinite programming. If the degree bound is chosen to be large enough, then the* **SDP***s will be feasible, and the certificates can be obtained from its solution.*

Algorithm 2 is an implementation of Theorem 2 and we will invoke Algorithm 2 as a subroutine later. Note that Algorithm 2 is a little different from the original one in [24], as here we require that f has 1 as a summand for our specific purpose.

4 Synthesizing Non-linear Interpolants in General Case

As discussed before, we only need to consider how to synthesize interpolants for the following two specific SASs

$$\mathcal{T}_1 = \begin{cases} f_1(\mathbf{x}) \geq 0, \ldots, f_{s_1}(\mathbf{x}) \geq 0, \\ g_1(\mathbf{x}) \neq 0, \ldots, g_{t_1}(\mathbf{x}) \neq 0, \\ h_1(\mathbf{x}) = 0, \ldots, h_{u_1}(\mathbf{x}) = 0 \end{cases} \quad \mathcal{T}_2 = \begin{cases} f_{s_1+1}(\mathbf{x}) \geq 0, \ldots, f_s(\mathbf{x}) \geq 0, \\ g_{t_1+1}(\mathbf{x}) \neq 0, \ldots, g_t(\mathbf{x}) \neq 0, \\ h_{u_1+l}(\mathbf{x}) = 0, \ldots, h_u(\mathbf{x}) = 0 \end{cases} \tag{3}$$

where $\mathcal{T}_1$ and $\mathcal{T}_2$ do not share any real solutions.

By Theorems 1&2, there exist $f \in \mathcal{C}(\{f_1, \ldots, f_s\})$, $g \in \mathit{Mult}(\{g_1, \ldots, g_t\})$ and $h \in \mathcal{I}(\{h_1, \ldots, h_u\})$ such that $f + g^2 + h \equiv 0$, where

$$\begin{aligned} g &= \Pi_{i=1}^{t} g_i^{2m}, \\ h &= q_1 h_1 + \cdots + q_{u_1} h_{u_1} + \cdots + q_u h_u, \\ f &= p_0 + p_1 f_1 + \cdots + p_s f_s + p_{12} f_1 f_2 + \cdots + p_{1\ldots s} f_1 \ldots f_s. \end{aligned}$$

in which q_i and p_i are in $\mathbf{SOS}$.

If f can be represented by three parts: the first part is an $\mathbf{SOS}$ polynomial that is greater than 0, the second part is from $\mathcal{C}(\{f_1, \ldots, f_{s_1}\})$, and the last part is from $\mathcal{C}(\{f_{s_1+1}, \ldots, f_s\})$, i.e., $f = p_0 + \sum_{v \subseteq \{1,\ldots,s_1\}} p_v(\Pi_{i \in v} f_i) + \sum_{v \subseteq \{s_1+1,\ldots,s\}} p_v(\Pi_{i \in v} f_i)$, where $\forall \mathbf{x} \in \mathbb{R}^n . p_0(\mathbf{x}) > 0$ and $p_v \in \mathbf{SOS}$. Then let

$$\begin{aligned} f_{\mathcal{T}_1} &= \sum_{v \subseteq 1,\ldots,s_1} p_v \Pi_{i \in v} f_i, \qquad h_{\mathcal{T}_1} = q_1 h_1 + \cdots + q_{u_1} h_{u_1}, \\ f_{\mathcal{T}_2} &= \sum_{v \subseteq s_1+1,\ldots,s} p_v \Pi_{i \in v} f_i, \qquad h_{\mathcal{T}_2} = h - h_{\mathcal{T}_1}, \\ q &= f_{\mathcal{T}_1} + g^2 + h_{\mathcal{T}_1} + \frac{p_0}{2} = -(f_{\mathcal{T}_2} + h_{\mathcal{T}_2}) - \frac{p_0}{2}. \end{aligned}$$

Obviously, we have $\forall \mathbf{x} \in \mathcal{T}_1 . q(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in \mathcal{T}_2 . q(\mathbf{x}) < 0$. Thus, let $I = q(\mathbf{x}) > 0$. We have $\mathcal{T}_1 \models I$ and $I \wedge \mathcal{T}_2 \models \bot$.

Notice that because the requirement on f cannot be guaranteed in general, the above approach is not complete generally. We will discuss under which condition the requirement can be guaranteed in the next section. We implement the above method for synthesizing non-linear interpolants in general case by Algorithm 3.

Example 2. Consider

$$\mathcal{T}_1 = \begin{cases} x_1^2 + x_2^2 + x_3^2 - 2 \geq 0, \\ x_1 + x_2 + x_3 \neq 0, \\ 1.2x_1^2 + x_2^2 + x_1 x_3 = 0 \end{cases} \text{ and } \mathcal{T}_2 = \begin{cases} -3x_1^2 - 4x_2^3 - 10x_3^2 + 20 \geq 0, \\ 2x_1 + 3x_2 - 4x_3 \neq 0, \\ x_1^2 + x_2^2 - x_3 - 1 = 0 \end{cases}$$

Algorithm 3. `SN_Interpolants`

input : $\mathcal{T}_1$ and $\mathcal{T}_2$ of the form $(3), b$
output: An interpolant I or NULL

$g := \Pi_{k=1}^{t} g_k^2$
$g := g^{\lfloor \frac{b}{\deg(g)} \rfloor}$
$\{f_{t_1}\} := \{\Pi_{i \in v} f_i \text{ for } v \subseteq \{1, \ldots, s_1\}\}$;
$\{f_{t_2}\} := \{\Pi_{i \in v} f_i \text{ for } v \subseteq \{s_1 + 1, \ldots, s\}\}$;
sdp:=`Certificate_Generation`$(\{f_{t_1}\} \cup \{f_{t_2}\}, g, \{h_1, \ldots, h_u\}, b)$
if sdp $\equiv$ NULL **then**
  return NULL
else
  $I := \frac{1}{2} + \sum_{v \subseteq \{1,\ldots,s_1\}} p_v \Pi_{i \in v} f_i + q_1 h_1 + \cdots + q_{u_1} h_{u_1} + g > 0$;
  return I;
end

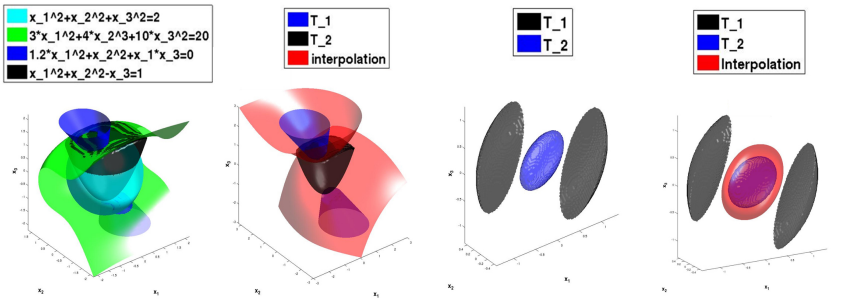


Fig. 1. Examples

Clearly, $\mathcal{T}_1$ and $\mathcal{T}_2$ do not share any real solutions, see Fig. 1 (the first part) [4]. By setting $b = 2$, after calling `Certificate_Generation`, we obtain an interpolant I with 30 monomials $-14629.26 + 2983.44x_3 + 10972.97x_3^2 + 297.62x_2 + 297.64x_2x_3 + 0.02x_2x_3^2 + 9625.61x_2^2 - 1161.80x_2^2x_3 + 0.01x_2^2x_3^2 + 811.93x_2^3 + 2745.14x_2^4 - 10648.11x_1 + 3101.42x_1x_3 + 8646.17x_1x_3^2 + 511.84x_1x_2 - 1034.31x_1x_2x_3 + 0.02x_1x_2x_3^2 + 9233.66x_1x_2^2 + 1342.55x_1x_2^2x_3 - 138.70x_1x_2^3 + 11476.61x_1^2 - 3737.70x_1^2x_3 + 4071.65x_1^2x_3^2 - 2153.00x_1^2x_2 + 373.14x_1^2x_2x_3 + 7616.18x_1^2x_2^2 + 8950.77x_1^3 + 1937.92x_1^3x_3 - 64.07x_1^3x_2 + 4827.25x_1^4 > 0$, whose figure is depicted in Fig. 1 (the second part). □

5 A Complete Algorithm under Archimedean Condition

Our approach to synthesizing non-linear interpolants presented in Section 4 is incomplete generally as it requires that the polynomial f in $\mathcal{C}(\{f_1, \ldots, f_s\})$ produced by Algorithm 2 can be represented by the sum of three polynomials, one of which is positive,

[4] For simplicity, we do not draw $x_1 + x_2 + x_3 \neq 0$, nor $2x_1 + 3x_2 - 4x_3 \neq 0$ in the figure.

the other two polynomials are from $\mathcal{C}(\{f_1,\ldots,f_{s_1}\})$ and $\mathcal{C}(\{f_{s_1+1},\ldots,f_s\})$ respectively. In this section, we show, under Archimedean condition, the requirement can be indeed guaranteed. Thus, our approach will become complete. In particular, we shall argue Archimedean condition is a necessary and reasonable restriction in practice.

5.1 Archimedean Condition

To the end, we need more knowledge of real algebraic geometry.

Definition 9 (quadratic module). *For $g_1,\ldots,g_m \in \mathbb{R}[\mathbf{x}]$, the set $\mathcal{M}(g_1,\ldots,g_m) = \{\delta_0 + \sum_{j=1}^{m} \delta_j g_j \mid \delta_0, \delta_j \in \mathcal{C}(\emptyset)\}$ is called the* quadratic module *generated by $g_1,\ldots,g_m$. A quadratic module $\mathcal{M}$ is called* proper *if $-1 \notin \mathcal{M}$ (i.e. $\mathcal{M} \neq \mathbb{R}[\mathbf{x}]$). A quadratic module $\mathcal{M}$ is* maximal *if for any $p \in \mathbb{R}[\mathbf{x}] \cap \overline{\mathcal{M}}$, $\mathcal{M} \cup \{p\}$ is not a quadratic module.*

In the sequel, we use $-\mathcal{M}$ to denote $\{-p \mid p \in \mathcal{M}\}$ for a given quadratic module $\mathcal{M}$.

The following results are adapted from [22,28] and will be used later, whose proofs can be found in [22,28].

Lemma 2 ([22,28]).

1) *If $\mathcal{M} \subseteq \mathbb{R}[\mathbf{x}]$ is a quadratic module, then $I = \mathcal{M} \cap -\mathcal{M}$ is an ideal.*
2) *If $\mathcal{M} \subseteq \mathbb{R}[\mathbf{x}]$ is a maximal proper quadratic module, then $\mathcal{M} \cup -\mathcal{M} = \mathbb{R}[\mathbf{x}]$.*
3) *$\{\mathbf{x} \in \mathbb{R}^n \mid f(\mathbf{x}) \geq 0\}$ is a compact set[5] for some $f \in \mathcal{M}(\{f_1,\ldots,f_s\})$ iff*

$$\forall p \in \mathbb{R}[\mathbf{x}], \exists n \in \mathbb{N}. n \pm p \in \mathcal{M}(f_1,\ldots,f_s). \tag{4}$$

Definition 10 (Archimedean). *For $g_1,\ldots,g_m \in \mathbb{R}[\mathbf{x}]$, the quadratic module $\mathcal{M}(g_1,\ldots,g_m)$ is said to be* Archimedean *if the condition (4) holds.*

$$\text{Let } \mathcal{T}_1 = f_1(\mathbf{x}) \geq 0,\ldots,f_{s_1}(\mathbf{x}) \geq 0 \text{ and } \mathcal{T}_2 = f_{s_1+1}(\mathbf{x}) \geq 0,\ldots,f_s(\mathbf{x}) \geq 0 \tag{5}$$

be two SASs, which contains constraints $c_l \leq x_i \leq c_r$ for every $x_i \in \mathbf{x}$, where c_l and c_r are reals, and $\mathcal{T}_1$ and $\mathcal{T}_2$ do not share real solutions.

Proposition 1. *Suppose $\{f_1(\mathbf{x}),\ldots,f_s(\mathbf{x})\}$ is given in (5), which contains constraints $c_l \leq x_i \leq c_r$ for every $x_i \in \mathbf{x}$. We can alway obtain a system $\{f_1(\mathbf{x}),\ldots,f_{s'}(\mathbf{x})\}$ such that $\mathcal{M}(f_1,\ldots,f_{s'})$ is Archimedean and $f_1 \geq 0 \wedge \cdots \wedge f_s \geq 0 \Leftrightarrow f_1 \geq 0 \wedge \cdots \wedge f_{s'} \geq 0$.*

Proof. For $\{f_1,\ldots,f_s\}$ in (5), as any variable is bounded, $N - \sum_{i=1}^{n} x_i^2 \geq 0$ is valid for some constant N under (5). We denote $N - \sum_{i=1}^{n} x_i^2$ by f_{s+1}. If $f_{s+1} \in \{f_1,\ldots,f_s\}$, then $\{f_1,\ldots,f_s\}$ is *Archimedean*. Otherwise, $\mathcal{M}(f_1,\ldots,f_s,f_{s+1})$ is *Archimedean*. □

Lemma 3. *[22,28] Let $\mathcal{M} \subseteq \mathbb{R}[\mathbf{x}]$ be a maximal and proper quadratic module, which is* Archimedean, *$I = \mathcal{M} \cap -\mathcal{M}$, and $f \in \mathbb{R}[\mathbf{x}]$, then there exists $a \in \mathbb{R}$ such that $f - a \in I$.*

Lemma 4. *If I is an ideal and there exists $\mathbf{a} = (a_1,\ldots,a_n) \in \mathbb{R}^n$ such that $x_i - a_i \in I$ for $i = 1,\ldots,n$, then for any $f \in \mathbb{R}[\mathbf{x}]$, $f - f(\mathbf{a}) \in I$.*

[5] S is a compact set in $\mathbb{R}^n$ iff S is a bounded closed set.

Proof. Because $x_i - a_i \in I$ for $i = 1, \ldots, n$, $\langle x_1 - a_1, \ldots, x_n - a_n \rangle \subseteq I$. For any $f \in \mathbb{R}[\mathbf{x}]$, $\langle x_1 - a_1, \ldots, x_n - a_n \rangle$ is a radical ideal[6] and $(f - f(\mathbf{a}))(\mathbf{a}) = 0$, so $f - f(\mathbf{a}) \in \langle x_1 - a_1, \ldots, x_n - a_n \rangle \subseteq I$. □

Theorem 3. *Suppose* $\{f_1(\mathbf{x}), \ldots, f_{s'}(\mathbf{x})\}$ *is given in Proposition (1). If* $\bigwedge_{i=1}^{s'}(f_i \geq 0)$ *is unsatisfiable i.e.* $\bigwedge_{i=1}^{s}(f_i \geq 0)$ *is unsatisfiable, then* $-1 \in \mathcal{M}(f_1, \ldots, f_{s'})$.

Proof. By Proposition 1, we only need to prove that the quadratic module $\mathcal{M}(f_1, \ldots, f_{s'})$ is not proper.

Assume $\mathcal{M}(f_1, \ldots, f_{s'})$ is proper. By Zorn's lemma, we can extend $\mathcal{M}(f_1, \ldots, f_{s'})$ to a maximal proper quadratic module $\mathcal{M} \supseteq \mathcal{M}(f_1, \ldots, f_{s'})$. Since $\mathcal{M}(f_1, \ldots, f_{s'})$ is Archimedean, $\mathcal{M}$ is also Archimedean. By Lemma 3, there exists $\mathbf{a} = (a_1, \ldots, a_n) \in \mathbb{R}^n$ such that $x_i - a_i \in I = \mathcal{M} \cap -\mathcal{M}$ for all $i \in \{1, \ldots, n\}$. From Lemma 4, $f - f(\mathbf{a}) \in I$ for any $f \in \mathbb{R}[\mathbf{x}]$. In particular, for $f = f_j$, we have $f_j(\mathbf{a}) = f_j - (f_j - f_j(\mathbf{a})) \in \mathcal{M}$, as $f_j \in \mathcal{M}(f_1, \ldots, f_{s'}) \subseteq \mathcal{M}$ and $-(f_j - f_j(\mathbf{a})) \in \mathcal{M}$, for $j = 1, \ldots, s'$. Suppose $f_j(\mathbf{a}) < 0$, then there exists $y \in \mathbb{R}$ such that $y^2 f_j \mathbf{a} = -1 \in \mathcal{M}$, which contradicts to the assumption, so $f_j(\mathbf{a}) \geq 0$. This contradicts to the unsatisfiability of $\bigwedge_{i=1}^{s'}(f_i \geq 0)$. □

By Theorem 3 we have $-1 \in \mathcal{M}(f_1, \ldots, f_{s'})$. So, there exist $\sigma_0, \ldots, \sigma_{s'} \in \mathcal{C}(\emptyset)$ such that $-1 = \sigma_0 + \sigma_1 f_1 + \cdots + \sigma_{s_1} f_{s_1} + \sigma_{s_1+1} f_{s_1+1} + \cdots + f_{s'} \sigma_{s'}$. It follows

$$-(\frac{1}{2} + \sigma_{s_1+1} f_{s_1+1} + \cdots + \sigma_{s'} f_{s'}) = \frac{1}{2} + \sigma_0 + \sigma_1 f_1 + \cdots + \sigma_{s_1} f_{s_1}. \tag{6}$$

Let $q(\mathbf{x}) = \frac{1}{2} + \sigma_0 + \sigma_1 f_1 + \cdots + \sigma_{s_1} f_{s_1}$, we hence have $\forall \mathbf{x} \in \mathcal{T}_1 . q(\mathbf{x}) > 0$ and $\forall \mathbf{x} \in \mathcal{T}_2 . f_{s'}(\mathbf{x}) \geq 0 \wedge q(\mathbf{x}) < 0$. So, let $I = q(\mathbf{x}) > 0$. According to Definition 1, I is an interpolant of $\mathcal{T}_1$ and $\mathcal{T}_2$. So, under Archimedean condition, we can revise Algorithm 3 as Algorithm 4.

Algorithm 4. `RSN_Interpolants`

input : $\mathcal{T}_1$ and T_2 as in (5)
output: I

```
1  b=2;
2  while true do
3  |  sdp=Certificate_Generation({f_1,...,f_s},0,{},b);
4  |  if sdp ≠ NULL then
5  |  |  I = {1/2 + Σ_{i=1}^{s_1} p_i f_i > 0};
6  |  |  return I;
7  |  else
8  |  |  b=b+2;
9  |  end
10 end
```

[6] Ideal I is a radical ideal if $I = \sqrt{I} = \{f | f^k \in I \text{ for some integer } k > 0\}$.

Example 3. Let $\Psi = \bigwedge_{i=1}^{3} x_i \geq -2 \wedge -x_i \geq -2$, $f_1 = -x_1^2 - 4x_2^2 - x_3^2 + 2$, $f_2 = x_1^2 - x_2^2 - x_1x_3 - 1$, $f_3 = -x_1^2 - 4x_2^2 - x_3^2 + 3x_1x_2 + 0.2$, $f_4 = -x_1^2 + x_2^2 + x_1x_3 + 1$. Consider $\mathcal{T}_1 = \Psi \wedge f_1 \geq 0 \wedge f_2 \geq 0$ and $\mathcal{T}_2 = \Psi \wedge f_3 \geq 0 \wedge f_4 \geq 0$. Obviously, $\mathcal{T}_1 \wedge T_2$ is unsatisfiable, see Fig. 1 (the third part).

By applying `RSN_Interpolants`, we can get an interpolant as $-33.7255x_1^4 + 61.1309x_1^3x_2 + 4.6818x_1^3x_3 - 57.927x_1^2x_2^2 + 13.4887x_1^2x_2x_3 - 48.9983x_1^2x_3^2 - 8.144x_1^2 - 48.1049x_1x_2^3 - 6.7143x_1x_2^2x_3 + 29.8951x_1x_2x_3^2 + 61.5932x_1x_2 + 0.051659x_1x_3^3 - 0.88593x_1x_3 - 34.7211x_2^4 - 7.8128x_2^3x_3 - 71.9085x_2^2x_3^2 - 60.5361x_2^2 - 1.6845x_2x_3^3 - 0.5856x_2x_3 - 15.2929x_3^4 - 9.7563x_3^2 + 6.7326 > 0$, which is depicted in Fig 1 (the fourth part). In this example, the final value of b is 2. □

5.2 Discussions

1. Reasonability of the Archimedean Condition: Only bounded numbers can be represented in computer, so it is reasonable to constraint each variable with upper and lower bounds in practice. Not allowing strict inequalities indeed reduces the expressiveness from a theoretical point of view. However, as only numbers with finite precision can be represented in computer, we can always relax a strict inequality to an equivalent non-strict inequality in practice too. In a word, we believe the *Archimedean condition* is reasonable in practice.

2. Necessity of the Archimedean Condition: In Theorem 3, the *Archimedean condition* is necessary. For example, let $\mathcal{T}_1 = \{x_1 \geq 0, x_2 \geq 0\}$ and $\mathcal{T}_2 = \{-x_1x_2 - 1 \geq 0\}$. So, $\mathcal{T}_1 \wedge \mathcal{T}_2 = \emptyset$ is not *Archimedean* and unsatisfiable, but $-1 \notin \mathcal{M}(x_1, x_2, -x_1x_2 - 1)$ (refer to the full version [27] for the proof).

6 Correctness and Complexity Analysis

The correctness of the algorithm `SN_Interpolants` is obvious according to Theorem 2 and the discussion of Section 4. Its complexity just corresponds to one iteration of the algorithm `RSN_Interpolants`. The correctness of the algorithm `RSN_Interpolants` is guaranteed by Theorem 2 and Theorem 3. The cost of each iteration of `RSN_Interpolants` depends on the number of the variables n, the number of polynomial constraints u, and the current highest degree d. The size of X in (1) is $u\binom{n+d/2}{n}$ and the m in (1) is $\binom{n+d}{n}$. So, the complexity of applying interior point method to solve the **SDP** is polynomial in $u\binom{n+d/2}{n}\binom{n+d}{n}$. Hence, the cost of each iteration of `RSN_Interpolants` is $u\binom{n+d/2}{n}\binom{n+d}{n}$. Therefore, the total cost of `RSN_Interpolants` is polynomial in $du\binom{n+d/2}{n}\binom{n+d}{n}$ for a given upper bound of degree d. For a given problem in which n, u are fixed, the complexity of the algorithm becomes polynomial in d. As indicated in [24], the theoretical bound of d is at least triply exponential. So our method can not solve every possible instances in polynomial time. The complexity of Algorithm `SN_Interpolants` is the same as above discussions, except that the number of polynomial constraints is about $2^{s_1} + 2^{s-s_1}$.

7 Implementation and Experimental Results

We have implemented a prototypical tool of the algorithms described in this paper, called `AiSat`, which contains 6000 lines of `C++` codes. `AiSat` calls `Singular` [29] to deal with polynomial input and `CSDP` to solve **SDP**s. In `AiSat`, we design a specific algorithm to transform polynomial constraints to matrix constraints, which indeed improves the efficiency of our tool very much, indicated by the comparison with `SOSTOOLS` [30] (see the table below). As a future work, we plan to implement a new **SDP** solver with more stability and convergence efficiency for solving **SDP**s.

In the following, we report the experimental results by applying `AiSat` to some benchmarks.

The first example is from [31], see the source code in Code 1.2. We show its correctness by applying `AiSat` to the following two possible executions.

- Subproblem 1: Suppose there is an execution starting from a state satisfying the assertion at line 13 (obviously, the initial state satisfies the assertion), after $\rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 9 \rightarrow 11 \rightarrow 12 \rightarrow 13$, ending at a state that does not satisfy the assertion.
  Then the interpolant synthesized by our approach is $716.77 + 1326.74ya + 1.33ya^2 + 433.90ya^3 + 668.16xa - 155.86xa{\cdot}ya + 317.29xa{\cdot}ya^2 + 222.00xa^2 + 592.39xa^2{\cdot}ya + 271.11xa^3 > 0$, which guarantees that this execution is infeasible.
- Subproblem 2 : Assume there is an execution starting from a state satisfying the assertion at line 13, after $\rightarrow 6 \rightarrow 7 \rightarrow 8 \rightarrow 10 \rightarrow 11 \rightarrow 12 \rightarrow 13$, ending at a state that does not satisfy the assertion.
  The interpolant generated by our approach is $716.95 + 1330.91ya + 67.78ya^2 + 551.51ya^3 + 660.66xa - 255.52xa{\cdot}ya + 199.84xa{\cdot}ya^2 + 155.63xa^2 + 386.87xa^2{\cdot}ya + 212.41xa^3 > 0$, which guarantees this execution is infeasible either.

```
1  int main ()
2  {   int x, y;
3      int xa := 0;
4      int ya := 0;
5      while (nondet())
6      {   x := xa + 2 * ya;
7          y := −2 * xa + ya;
8          x++;
9          if (nondet()) y = y + x;
10         else y := y − x;
11         xa := x − 2 * y;
12         ya := 2 * x + y;   }
13     assert (xa + 2 * ya >= 0);
14     return 0;   }
```

Code 1.2:ex1

```
1  vc := 0;
2    /* the initial veclocity */
3  fr := 1000;
4    /* the initial force */
5  ac := 0.0005 * fr;
6     /* the initial acceleration */
7  while ( 1 )
8  {   fa := 0.5418 * vc * vc;
9        /* the force control */
10   fr := 1000 − fa;
11   ac := 0.0005 * fr;
12   vc := vc + ac;
13    assert(vc < 49.61);
14       /* the safety velocity */ }
```

Code 1.3: An accelerating car

The second example `accelerate` (see Code 1.3) is from [21]. Taking the air resistance into account, the relation between the car's velocity and the physical drag contains quadratic functions. Due to air resistance the velocity of the car cannot surpass

$49.61m/s$, which is a safety property. Assume that there is an execution $(vc < 49.61) \to 8 \to 10 \to 11 \to 12 \to 13(vc \geq 49.61)$. By applying `AiSat`, we can obtain an interpolant $-1.3983vc + 69.358 > 0$, which guarantees $vc < 49.61$. So, `accelerate` is correct.

The last example `logistic` is also from [21]. Mathematically, the logistic loop is written as $x_{n+1} = rx_n(1 - x_n)$, where $0 \leq x_n \leq 1$. When $r = 3.2$, the logistic loop oscillates between two values. The verification obligation is to guarantee that it is within the safe region $(0.79 \leq x \leq 0.81) \vee (0.49 \leq x \leq 0.51)$. By applying `AiSat` to the following four possible executions, the correctness is obtained.

- Subproblem 1: $\{x \geq 0.79 \wedge x \leq 0.81\}$ `logistic` $\{x > 0.51\}$ is invalidated by the synthesized interpolant $108.92 - 214.56x > 0$.
- Subproblem 2: $\{x \geq 0.79 \wedge x \leq 0.81\}$ `logistic` $\{x < 0.49\}$ is outlawed by the synthesized interpolant $-349.86 + 712.97x > 0$.
- Subproblem 3: $\{x \geq 0.49 \wedge x \leq 0.51\}$ `logistic` $\{x > 0.81\}$ is excluded by the generated interpolant $177.21 - 219.40x > 0$.
- Subproblem 4: $\{x \geq 0.49 \wedge x \leq 0.51\}$ `logistic` $\{x < 0.79\}$ is denied by the generated interpolant $-244.85 + 309.31x > 0$.

The experimental results of applying `AiSat` to the above three examples on a desktop (64-bit Intel(R) Core(TM) i5 CPU 650 @ 3.20GHz, 4GB RAM memory and Ubuntu 12.04 GNU/Linux) are listed in the table below. Meanwhile, as a comparison, we apply the SOSTOOLS to the three examples with the same computer.

Benchmark	#Subporblems	`AiSat` (milliseconds)	`SOSTOOLS` (milliseconds)
`ex1`	2	60	3229
`accelerate`	1	940	879
`logistic`	4	20	761

8 Conclusion

The main contributions of the paper include:

- We give a sound but inomplete algorithm `SN_Interpolants` for the generation of interpolants for non-linear arithmetic in general.
- If the two systems satisfy *Archimedean condition*, we provide a more practical algorithm `RSN_Interpolants`, which is not only sound but also complete, for generating Craig interpolants.
- We implement the above algorithms as a protypical tool `AiSat`, and demonstrate our approach by applying the tool to some benchmarks.

In the future, we will focus how to relax the Archimedean condition and how to combine non-linear arithmetic with other well-established decidable first order theories. In particular, we believe that we can use the method of [32,21] to extend our algorithm to uninterpreted functions. To investigate errors caused by numerical computation in **SDP** is quite interesting. In addition, it deserves to investigate the possibility to apply our results to verify hybrid systems.

References

1. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Logic of Programs, vol. 131. pp. 52–71 (1981)
2. Nipkow, T., Wenzel, M., Paulson, L.C.: Isabelle/HOL: a proof assistant for higher-order logic. Springer, Heidelberg (2002)
3. Owre, S., Rushby, J., Shankar, N.: PVS: A prototype verification system. In: Kapur, D. (ed.) CADE 1992. LNCS, vol. 607, pp. 748–752. Springer, Heidelberg (1992)
4. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977, pp. 238–252 (1977)
5. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: Cleaveland, W.R. (ed.) TACAS 1999. LNCS, vol. 1579, pp. 193–207. Springer, Heidelberg (1999)
6. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract DPLL procedure to DPLL(T). J. ACM 53(6), 937–977
8. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
9. Craig, W.: Linear reasoning: A new form of the Herbrand-Gentzen theorem. J. Symb. Log. 22(3), 250–268 (1957)
10. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM 5(7), 394–397 (1962)
11. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
12. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
13. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121
14. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
15. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL 2004, pp. 232–244 (2004)
16. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
17. Jung, Y., Lee, W., Wang, B.-Y., Yi, K.: Predicate generation for learning-based quantifier-free loop invariant inference. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 205–219. Springer, Heidelberg (2011)
18. Yorsh, G., Musuvathi, M.: A combination method for generating interpolants. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 353–368. Springer, Heidelberg (2005)
19. Kapur, D., Majumdar, R., Zarba, C.: Interpolation for data structures. In: FSE 2006, pp. 105–116 (2006)
20. Rybalchenko, A., Sofronie-Stokkermans, V.: Constraint solving for interpolation. J. Symb. Comput. 45(11), 1212–1233 (2010)
21. Kupferschmid, S., Becker, B.: Craig interpolation in the presence of non-linear constraints. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 240–255. Springer, Heidelberg (2011)
22. Bochnak, J., Coste, M., Roy, M.F.: Real Algebraic Geometry. Springer (1998)

23. Parrilo, P.A.: Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization. PhD thesis, California Inst. of Tech. (2000)
24. Parrilo, P.A.: Semidefinite programming relaxations for semialgebraic problems. Mathematical Programming 96, 293–320 (2003)
25. Vandenberghe, L., Boyd, S.: Semidefinite programming. SIAM Review 38(1), 49–95 (1996)
26. Chen, Y., Xia, B., Yang, L., Zhan, N.: Generating polynomial invariants with DISCOVERER and QEPCAD. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) Formal Methods and Hybrid Real-Time Systems. LNCS, vol. 4700, pp. 67–82. Springer, Heidelberg (2007)
27. Dai, L., Xia, B., Zhan, N.: Generating non-linear interpolants by semidefinite programming. CoRR abs/1302.4739 (2013)
28. Laurent, M.: Sums of squares, moment matrices and optimization over polynomials. In: Emerging Applications of Algebraic Geometry. The IMA Volumes in Mathematics and its Applications, vol. 149, pp. 157–270 (2009)
29. Greuel, G.M., Pfister, G., Schönemann, H.: Singular: a computer algebra system for polynomial computations. ACM Commun. Comput. Algebra 42(3), 180–181 (2009)
30. Prajna, S., Papachristodoulou, A., Seiler, P., Parrilo, P.A.: SOSTOOLS: Sum of squares optimization toolbox for MATLAB (2004)
31. Gulavani, B., Chakraborty, S., Nori, A., Rajamani, S.: Automatically refining abstract interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
32. Sofronie-Stokkermans, V.: Interpolation in local theory extensions. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 235–250. Springer, Heidelberg (2006)

Under-Approximating Loops in C Programs for Fast Counterexample Detection*

Daniel Kroening[1], Matt Lewis[1], and Georg Weissenbacher[2,**]

[1] Oxford University
[2] Vienna University of Technology, Austria

Abstract. Many software model checkers only detect counterexamples with deep loops after exploring numerous spurious and increasingly longer counterexamples. We propose a technique that aims at eliminating this weakness by constructing auxiliary paths that represent the effect of a range of loop iterations. Unlike acceleration, which captures the exact effect of arbitrarily many loop iterations, these auxiliary paths may under-approximate the behaviour of the loops. In return, the approximation is sound with respect to the bit-vector semantics of programs.

Our approach supports arbitrary conditions and assignments to arrays in the loop body, but may as a result introduce quantified conditionals. To reduce the resulting performance penalty, we present two quantifier elimination techniques specially geared towards our application.

Loop under-approximation can be combined with a broad range of verification techniques. We paired our techniques with lazy abstraction and bounded model checking, and evaluated the resulting tool on a number of buffer overflow benchmarks, demonstrating its ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

1 Introduction

The generation of *diagnostic counterexamples* is a key feature of model checking. Counterexamples serve as witness for the refutation of a property, and are an invaluable aid to the engineer for understanding and repairing the fault.

Counterexamples are particularly important in software model checking, as bugs in software frequently require thousands of transitions to be executed, and are thus difficult to reproduce without the help of an explicit error trace. Existing software model checkers, however, fail to scale when analysing programs with bugs that involve many iterations of a loop. The primary reason for the inability of many existing tools to discover such "deep" bugs is that exploration is performed in a breadth-first fashion: the detection of an unsafe execution traversing a loop involves the repeated refutation of increasingly longer spurious

* Supported by the Engineering and Physical Sciences Research Council (EPSRC) under grant no. EP/H017585/1, the EU FP7 STREP PINCETTE, the ARTEMIS VeTeSS project, and ERC project 280053.

** Funded by the Vienna Science and Technology Fund (WWTF) through project VRG11-005 and the Austrian Science Fund (FWF) through RiSE (S11403-N23).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 381–396, 2013.

counterexamples. The analyser first considers a potential error trace with one loop iteration, only to discover that this trace is infeasible. As consequence, the analyser will increase the search depth, usually by considering one further loop iteration. In practice, the computational effort required to discover an assertion violation thus grows exponentially with the depth of the bug.

Notably, the problem is not limited to procedures based on abstraction, such as predicate abstraction or abstraction with interpolants. Bounded Model Checking (BMC) is optimised for discovering bugs up to a given depth k, but the computational cost grows exponentially in k.

The contribution of this paper is a new technique that enables scalable detection of deep bugs. We transform the program by adding a new, auxiliary path to loops that summarises the effect of a parametric number of iterations of the loop. Similar to acceleration, which captures the exact effect of arbitrarily many iterations of an integer relation by computing its reflexive transitive closure in one step [3,6,9], we construct a summary of the behaviour of the loop. By symbolically bounding the number of iterations, we obtain an *under-approximation* which is sound with respect to the bit-vector semantics of programs. Thus, we avoid false alarms that might be triggered by modeling variables as integers.

In contrast to related work, our technique supports assignments to arrays and arbitrary conditional branching by computing quantified conditionals. As the computational cost of analysing programs with quantifiers is high, we introduce two novel techniques for summarising certain conditionals without quantifiers. The key insight is that many conditionals in programs (e.g., loop exit conditions such as $\texttt{i} \leq 100$ or even $\texttt{i} \neq 100$) exhibit a certain monotonicity property that allows us to drop quantifiers.

Our approximation can be combined soundly with a broad range of verification engines, including predicate abstraction, lazy abstraction with interpolation [16], and bounded software model checking [4]. To demonstrate this versatility, we combined our technique with lazy abstraction and the CBMC [4] model checker. We evaluated the resulting tool on a large suite of benchmarks known to contain deep paths, demonstrating our ability to efficiently detect deep counterexamples in C programs that manipulate arrays.

2 Outline

2.1 Notation and Preliminaries

We restrict our presentation to a simple imperative language comprising assignments, assumptions, and assertions. A program is a control flow graph $\langle V, E, \lambda \rangle$, where V and E are sets of vertices and edges, respectively, and λ is a labelling function mapping vertices to statements. Procedure calls are in-lined and omitted in our presentation. The behaviour of a program is defined by the paths in the control flow graph (CFG). A path π of length m is a sequence of contiguous edges $e_1\, e_2 \ldots e_m$ ($e_i \in E$, $1 \leq i \leq m$). Abusing our notation, we use the corresponding sequence of statements $\lambda(e_1); \lambda(e_1); \ldots \lambda(e_m)$ to represent paths

Table 1. Predicate transformers for simple program statements and paths. $Q[\mathtt{x}/e]$ denotes that all free occurrences of $\mathtt{x}$ in Q are replaced with the expression e.

Path π	**Strongest Postcondition** $sp(\pi, P)$	**Weakest Liberal Precondition** $wlp(\pi, Q)$
ε / `skip`	P	Q
`x:=`e	$\exists \mathtt{`x} \,.\, (\mathtt{x} = e[\mathtt{x}/\mathtt{`x}]) \wedge P[\mathtt{x}/\mathtt{`x}]$	$Q[\mathtt{x}/e]$
$[R]$	$P \wedge R$	$R \Rightarrow Q$
`assert(`R`)`	$P \wedge R$	$R \Rightarrow Q$
$\pi_1 \,;\, \pi_2$	$sp(\pi_2, sp(\pi_1, P))$	$wlp(\pi_1, wlp(\pi_2, Q))$
$\pi_1 \sqcap \pi_2$	$sp(\pi_1, P) \vee sp(\pi_2, P)$	$wlp(\pi_1, Q) \wedge wlp(\pi_2, Q)$

(where ; denotes the non-commutative path concatenation operator). We use ε to denote the path of length 0 and inductively define π^n as $\pi^0 = \varepsilon$ and $\pi^{n+1} = \pi^n; \pi$ (for $n \geq 0$). In accordance with [17], $\pi_1 \sqcap \pi_2$ represents the non-deterministic choice between two paths, i.e., $\overset{\pi_1}{\underset{\pi_2}{\diamond\!\!\rightrightarrows}}$. The commutative operator $\sqcap$ is extended to sets of paths in the usual manner.

We use first-order logic (defined as usual) with background theories commonly used in software verification (such as arithmetic, bit-vectors, arrays and uninterpreted functions) to represent program expressions and predicates. T (F) represents the predicate that is always true (false). We use $*$ to indicate non-deterministic values. The semantics for statements and paths is determined by the predicate transformers in Table 1 (see [17]). A Hoare triple $\{P\}\ \pi\ \{Q\}$ comprises a pre-condition P, a path π, and a post-condition Q such that $sp(\pi, P)$ implies Q. Given a set $\mathtt{X} = \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of k variables, we introduce corresponding sets $\mathtt{`X} = \{\mathtt{`x}_1, \ldots, \mathtt{`x}_k\}$ and $\mathtt{X'} = \{\mathtt{x}'_1, \ldots, \mathtt{x}'_k\}$ of *primed* variables to refer to variables in prior and subsequent time-frames, respectively (where the term *time-frame* refers to an instance of π in π^n). We use $\mathtt{X}^{\langle i \rangle} = \{\mathtt{x}_1^{\langle i \rangle}, \ldots, \mathtt{x}_k^{\langle i \rangle}\}$ to refer to the variables in a specific time-frame i. The transition relation of π is the predicate $\neg wlp(\pi, \bigvee_{i=1}^{k} \mathtt{x}_i \neq \mathtt{x}'_i)$ [17] and relates variables of two time frames (for example, for $k = 2$ and the path $\pi = [\mathtt{x}_1 < 0]; \mathtt{x}_1 = \mathtt{x}_2 + 1$ we obtain $(\mathtt{x}_1 < 0) \wedge (\mathtt{x}'_1 = \mathtt{x}_2 + 1) \wedge (x'_2 = \mathtt{x}_2)$).

2.2 A Motivating Example

A common characteristic of many contemporary symbolic software model checking techniques (such as counterexample-guided abstraction refinement with predicate abstraction [1,8], lazy abstraction with interpolants [16], and bounded model checking [4]) is that the computational effort required to discover an assertion violation may increase exponentially with the length of the corresponding counterexample path (c.f. [13]). In particular, the detection of assertion violations that require a large number of loop iterations results in the enumeration of increasingly longer *spurious* counterexamples traversing that loop. This problem is illustrated by the following example.

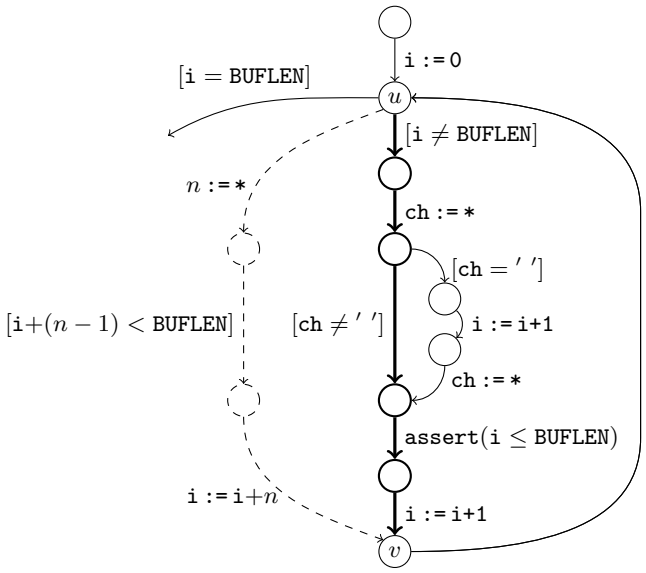


Fig. 1. CFG with path π (bold) and approximated path $\overrightarrow{\pi}$ (dashed)

Example 1. Figure 1 shows a program fragment derived from code permitting a buffer overflow (detected by the assertion) to occur in the n^{th} iteration of the loop if `i` reaches (`BUFLEN` -1) and the branch $[\texttt{ch} = \texttt{' '}]$ is taken in the $(n-1)^{\text{th}}$ iteration. The verification techniques mentioned above explore the paths in order of increasing length. The shortest path that reaches the assertion does not violate it, as

$$sp((\texttt{i} := 0; [\texttt{i} \neq \texttt{BUFLEN}\}]; \texttt{ch} := *; [\texttt{ch} \neq \texttt{' '}]), \mathsf{T}) \quad \Rightarrow \quad (\texttt{i} \leq \texttt{BUFLEN})\,.$$

In a predicate abstraction or lazy abstraction framework, this path represents the first in a series of spurious counterexamples of increasing length. Let π denote the path emphasised in Figure 1, which traverses the loop once. The verification tool will generate a family of spurious counterexamples with the prefixes $\texttt{i := 0;}\ \pi^n$ (where $0 < n \leq \frac{\texttt{BUFLEN}}{2}$) before it detects a path long enough to violate the assertion. Each of these paths triggers a computationally expensive refinement cycle. Similarly, a bounded model checker will fail to detect a counterexample unless the loop bound is increased to $\frac{\texttt{BUFLEN}}{2} + 1$.

The iterative exploration of increasingly deeper loops primarily delays the detection of assertion violations (c.f. [13]), but can also result in a diverging series of interpolants and predicates if the program is safe (see [10]).

2.3 Approximating Paths with Loops

We propose a technique that aims at avoiding the enumeration of paths with an insufficient number of loop iterations. Our approach is based on the insight that

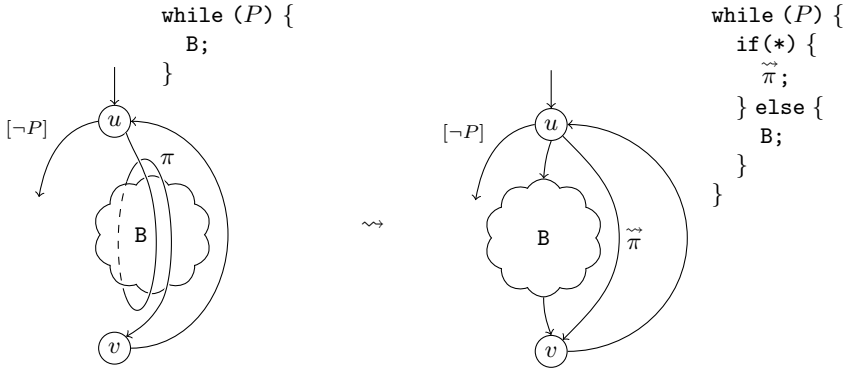


Fig. 2. Approximating the natural loop with head u and back-edge $v \to u$. Path π is a path traversing the body `B` at least once, and may take different branches in `B` in subsequent iterations.

the refutation of spurious counterexamples containing a sub-path of the form π^n is futile if there exists an n large enough to permit an assertion violation. We add an auxiliary path that bypasses the original loop body and represents the effect of π^n for a range of n (detailed later in the paper). Our approach comprises the following steps:

1. We sensitise an existing tool to detect paths π that repeatedly traverse the loop body `B` (as illustrated in the left half of Figure 2). We emphasise that π may span more than one iteration of the loop, and that the branches of `B` taken by π in different iterations may vary.
2. We construct a path $\overset{\rightsquigarrow}{\pi}$ whose behaviour *under-approximates* $\bigsqcap\{\pi^n \mid n \geq 0\}$. This construction does not correspond to acceleration in a strict sense, since $\overset{\rightsquigarrow}{\pi}$ (as an under-approximation) does not necessarily represent an arbitrary number of loop iterations. §3 describes techniques to derive $\overset{\rightsquigarrow}{\pi}$.
3. By construction, the assumptions in $\overset{\rightsquigarrow}{\pi}$ may contain universal quantifiers ranging over an auxiliary variable which encodes the number of loop iterations. In §4, we discuss two cases in which (some of) these quantifiers can be eliminated, namely (a) if the characteristic function of the predicate $\neg wlp(\pi^n, \mathsf{F})$ is *monotonic* in the number of loop iterations n, or (b) if π^n modifies an array and the indices of the modified array elements can be characterised by means of a quantifier-free predicate. We show that in certain cases condition (a) can be met by splitting π into several separate paths.
4. We augment the control flow graph with an additional branch of the loop containing $\overset{\rightsquigarrow}{\pi}$ (Figure 2, right). §5 demonstrates empirically how this program transformation can accelerate the detection of bugs that require a large number of loop iterations.

The following example demonstrates how our technique accelerates the detection of the buffer overflow of Example 1.

Example 2. Assume that the verification tool encounters the node u in Figure 1 a second time during the exploration of a path (u is the head of a natural loop with back-edge $v \to u$). We conclude that there exists a family of (sub-)paths π^n induced by the number n of loop iterations. The repeated application of the strongest post-condition to the parametrised path π^n for an increasing n gives rise to a recurrence equation $\mathtt{i}^{\langle n \rangle} = \mathtt{i}^{\langle n-1 \rangle} + 1$ (for clarity, we work on a sliced path omitting statements referring to `ch`):

$$\begin{aligned}
sp(\pi^1, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle} \,.\, (\mathtt{i}^{\langle 0 \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i} = \mathtt{i}^{\langle 0 \rangle} + 1) \\
sp(\pi^2, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle}, \mathtt{i}^{\langle 1 \rangle} \,.\, (\mathtt{i}^{\langle 0 \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle 1 \rangle} < \mathtt{BUFLEN}) \wedge \\
&\qquad\qquad (\mathtt{i}^{\langle 1 \rangle} = \mathtt{i}^{\langle 0 \rangle} + 1) \wedge (\mathtt{i} = \mathtt{i}^{\langle 1 \rangle} + 1) \\
&\;\;\vdots \\
sp(\pi^n, \mathsf{T}) &= \exists \mathtt{i}^{\langle 0 \rangle} \ldots \mathtt{i}^{\langle n-1 \rangle} \,.\, \left(\textstyle\bigwedge_{j=0}^{n-1} (\mathtt{i}^{\langle j \rangle} < \mathtt{BUFLEN}) \wedge (\mathtt{i}^{\langle j+1 \rangle} = \mathtt{i}^{\langle j \rangle} + 1) \right)
\end{aligned}$$

where $\mathtt{i}^{\langle n \rangle}$ in the last line represents $\mathtt{i}$ after the execution of π^n. This recurrence equation can be put into its equivalent closed form $\mathtt{i}^{\langle n \rangle} = \mathtt{i}^{\langle 0 \rangle} + n$. By assigning n a (positive) non-deterministic value, we obtain the approximation (which happens to be exact in this case):

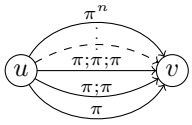
$$\overrightarrow{\overrightarrow{\pi}} \quad = \quad n := *; [\forall j \in [0, n) \,.\, \mathtt{i} + j < \mathtt{BUFLEN}]; \mathtt{i} := \mathtt{i} + n \quad .$$

Let us ignore arithmetic over- or under-flow for the time being (this topic is addressed in §3.4). We can then observe the following: if the predicate $\mathtt{i} + j < \mathtt{BUFLEN}$ is true for $j = n - 1$, then it must be true for any $j < n - 1$, i.e., the characteristic function of the predicate is *monotonic* in its parameter j. It is therefore possible to eliminate the universal quantifier and replace the assumption in $\overrightarrow{\overrightarrow{\pi}}$ with $(\mathtt{i} + (n-1) < \mathtt{BUFLEN})$. The dashed path in Figure 1 illustrates the corresponding modification of the original program. The resulting transformed program permits the violation of the assertion in the original loop body after a single iteration of $\overrightarrow{\overrightarrow{\pi}}$ (corresponding to `BUFLEN`-1 iterations of π).

The following presents techniques to compute the under-approximation $\overrightarrow{\overrightarrow{\pi}}$.

3 Under-Approximation Techniques

This section covers techniques to compute under-approximations $\overrightarrow{\overrightarrow{\pi}}$ of $\square\{\pi^n \mid n \geq 0\}$ such that $\overrightarrow{\overrightarrow{\pi}}$ is a condensation of the CFG fragment to the right.



The construction of $\overrightarrow{\overrightarrow{\pi}}$ has two aspects. Firstly, we need to make sure that all variables modified in $\overrightarrow{\overrightarrow{\pi}}$ are assigned values consistent with π^n for a non-deterministic choice of n. Secondly, $\overrightarrow{\overrightarrow{\pi}}$ must only allow choices of n for which $\neg wlp(\pi^n, \mathsf{F})$ is satisfiable, i.e., the corresponding path π^n must be *feasible*.

Our approximation technique is based on the observation that the sequence of assignments in π^n to a variable $\mathtt{x} \in \mathtt{X}$ corresponds to a recurrence equation (c.f. Example 2). The goal is to derive an equivalent *closed* form $\mathtt{x} := f_{\mathtt{x}}(\mathtt{X}, n)$. While there is a range of techniques to solve recurrence equations, we argue that it is sufficient to consider closed-form solutions that have the form of low-degree polynomials. The underlying argument is that a super-polynomial growth of variable values typically leads to an arithmetic overflow after a small number of iterations, which can be detected at low depth using conventional techniques.

The following sub-section focuses on deriving closed forms from a sequence of assignments to scalar integer variables, leaving conditionals aside. §3.2 covers assignments to arrays. Conditionals and path feasibility are addressed in §3.3. §3.4 addresses bit-vector semantics and arithmetic overflow.

3.1 Computing Closed Forms of Assignments

Syntactic Matching. A simple technique to derive closed forms is to check whether the given recurrence equation matches a pre-determined format. In our work on loop detection for predicate abstraction [13,14], we apply the following scheme:

$$\mathtt{x}^{\langle 0\rangle} = \alpha, \quad \mathtt{x}^{\langle n\rangle} = \mathtt{x}^{\langle n-1\rangle} + \beta + \gamma \cdot n \quad \rightsquigarrow \quad \mathtt{x}^{\langle n\rangle} = \alpha + \beta n + \gamma \frac{n \cdot (n+1)}{2}, \quad (1)$$

where $n > 0$ and α, β, and γ are numeric constants or loop-invariant *symbolic* expressions and $\mathtt{x}$ is the variant. This technique is computationally cheap and sufficient to construct the closed form $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle 0\rangle} + n$ of the recurrence equation $\mathtt{i}^{\langle n\rangle} = \mathtt{i}^{\langle n-1\rangle} + 1$ derived from the assignment $\mathtt{i} := \mathtt{i} + 1$ in Example 2.

Constraint-Based Acceleration. The disadvantage of a syntax-based approach is that it is limited to assignments following a simple pattern. Moreover, the technique is contingent on the syntax of the program fragment and may therefore fail even if there *exists* an appropriate polynomial representing the given assignments. In this section, we present an alternative technique that relies on a constraint solver to identify the coefficients of the polynomial $f_{\mathtt{x}}$.

Let $\mathtt{X}$ be the set $\{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ of variables in π. (In the following, we omit the braces $\{\}$ if clear from the context.) As previously, we start with the assumption that for each variable $\mathtt{x}$ modified in π, there is a low-degree polynomial in n

$$f_{\mathtt{x}}(\mathtt{X}^{\langle 0\rangle}, n) \stackrel{\text{def}}{=} \sum_{i=1}^{k} \alpha_i \cdot \mathtt{x}_i^{\langle 0\rangle} + \left(\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0\rangle} + \alpha_{(2\cdot k+1)} \right) \cdot n + \alpha_{(2\cdot k+2)} \cdot n^2 \quad (2)$$

over the initial variables $\mathtt{x}_1^{\langle 0\rangle}, \ldots, \mathtt{x}_k^{\langle 0\rangle}$ which accurately represents the value assigned to $\mathtt{x}$ in π^n (for $n \geq 1$). In other words, for each variable $\mathtt{x} \in \mathtt{X}$ modified in π, we assume that the following Hoare triple is valid:

$$\{ \bigwedge_{i=1}^{k} \text{`}\mathtt{x}_i = \mathtt{x}_i \}\ \pi^n\ \{ \mathtt{x} = f_{\mathtt{x}}(\text{`}\mathtt{x}_1, \ldots, \text{`}\mathtt{x}_k, n) \} \quad (3)$$

For each $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ we can generate $2 \cdot k + 2$ distinct assignments to $\mathtt{x}_1^{\langle 0 \rangle}, \ldots, \mathtt{x}_k^{\langle 0 \rangle}$, and n in (2) which determine a system of linearly independent equations over α_i, $0 < i \leq 2 \cdot k + 2$ (this can be proven by induction on k). If a solution to this system of equations exists, it *uniquely* determines the parameters $\alpha_1, \ldots, \alpha_{2 \cdot k+2}$ of the polynomial $f_{\mathtt{x}}$ for $\mathtt{x}$. In particular, the satisfiability of the encoding from which we derive the assignments guarantees that (3) holds for $0 \leq n \leq 2$. For larger values of n, we check the validity of (3) with respect to each $f_{\mathtt{x}}$ by means of induction. The validity of (3) follows (by induction over the length of the path π^n) from the validity of the base case established above, the formula (4) given below (which can be easily checked using a model checker or a constraint solver), and Hoare's rule of composition:

$$\{\bigwedge_{i=1}^{k} \left(\grave{\mathtt{x}}_i = \mathtt{x}_i\right) \wedge \mathtt{x} = f_{\mathtt{x}}(\grave{\mathtt{x}}_1, \ldots, \grave{\mathtt{x}}_k, n)\} \; \pi \; \{\mathtt{x} = f_{\mathtt{x}}(\grave{\mathtt{x}}_1, \ldots, \grave{\mathtt{x}}_k, n+1)\} \tag{4}$$

If for one or more $\mathtt{x} \in \{\mathtt{x}_1, \ldots, \mathtt{x}_k\}$ our technique fails to find valid parameters $\alpha_1, \ldots, \alpha_{2 \cdot k+2}$, or the validity check for $f_{\mathtt{x}}$ fails, we do not construct $\overset{\rightsquigarrow}{\pi}$.

Remark. The construction of under-approximations is not limited to the two techniques discussed above and can be based on other (and potentially more powerful) recurrence solvers.

3.2 Assignments to Arrays

Buffer overflows constitute a prominent class of safety violations that require a large number of loop iterations to surface. In C programs, buffers and strings are typically implemented using arrays. Let $\mathtt{i}$ be the variant of a loop which contains an assignment $\mathtt{a[i]} := e$ to an array $\mathtt{a}$. For a single iteration, we obtain

$$sp(\mathtt{a[i]} := e, P) \stackrel{\text{def}}{=} \exists \grave{\mathtt{a}} \,.\, \mathtt{a[i]} = e[\mathtt{a}/\grave{\mathtt{a}}] \wedge \forall j \neq \mathtt{i} \,.\, \left(\mathtt{a}[j] = \grave{\mathtt{a}}[j]\right) \wedge P[\mathtt{a}/\grave{\mathtt{a}}] \tag{5}$$

Assume further that closed forms for the variant $\mathtt{i}$ and the expression e exist (abusing our notation, we use f_e to refer to the latter). Given an initial pre-condition $P = \mathsf{T}$, we obtain the following *approximation*[1] after n iterations:

$$\forall j \in [0, n) \,.\, \mathtt{a}^{\langle n \rangle}[f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j)] = f_e(\mathtt{X}^{\langle 0 \rangle}, j) \wedge \\ \forall i \in \operatorname{dom} \mathtt{a} \,.\, \underbrace{\left(\exists j \in [0, n) \,.\, i = f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j)\right)}_{\text{membership test}} \vee \left(\mathtt{a}^{\langle n \rangle}[i] = \mathtt{a}^{\langle 0 \rangle}[i]\right), \tag{6}$$

where the domain ($\operatorname{dom} \mathtt{a}$) of $\mathtt{a}$ denotes the valid indices of the array. Notably, the membership test determining whether an array element is modified or not introduces quantifier alternation, posing a challenge to contemporary decision procedures. §4 addresses the elimination of the existential quantifier in (6).

[1] Condition (6) *under*-approximates the strongest post-condition, since there may exist $j_1, j_2 \in [0, n)$ such that $j_1 \neq j_2 \wedge f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j_1) = f_{\mathtt{i}}(\mathtt{X}^{\langle 0 \rangle}, j_2)$ and (6) is unsatisfiable. A similar situation arises if a loop body π contains multiple updates of the same array.

3.3 Assumptions and Feasibility of Paths

The techniques discussed in §3.1 yield polynomials and constraints representing the assignment statements of π^n, but leave aside the conditional statements which determine the feasibility of the path. In the following, we demonstrate how to derive the pre-condition $\neg wlp(\pi^n, \mathsf{F})$ using the polynomials $f_{\mathtt{x}}$ for $\mathtt{x} \in \mathtt{X}$.

Let $f_{\mathtt{X}}(\mathtt{X}, n) \stackrel{\text{def}}{=} \{f_{\mathtt{x}}(\mathtt{X}, n) \,|\, \mathtt{x} \in \mathtt{X}\}$ and let $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, n)]$ denote the simultaneous substitution of all free occurrences of the variables $\mathtt{x} \in \mathtt{X}$ in Q with the corresponding term $f_{\mathtt{x}}(\mathtt{X}, n)$.

Lemma 1. *The following equivalence holds:*

$$wlp(\pi^n, \mathsf{F}) \quad \equiv \quad \exists j \in [0, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]$$

Proof. Intuitively, the path π^n is infeasible if for *any* $j < n$ the first time-frame of the suffix $\pi^{(n-j)}$ is infeasible. We prove the claim by induction over n. Due to (3) and (4) we have $f_{\mathtt{X}}(\mathtt{X}, 0) = \mathtt{X}$ and $f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X}, n), 1) = f_{\mathtt{X}}(\mathtt{X}, n+1)$ (for $n \geq 0$).

Base case: $wlp(\pi, \mathsf{F}) \equiv \exists j \in [0, 0) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)] = \mathsf{F}$

Induction step. We start by applying the induction hypothesis:

$$\begin{aligned} wlp(\pi^n, \mathsf{F}) &\equiv wlp(\pi, \big(wlp(\pi^{n-1}, \mathsf{F})\big)) \\ &\equiv wlp(\pi, \exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]) \end{aligned}$$

We consider the effect of assignments and assumptions occurring in π on the post-condition $Q \stackrel{\text{def}}{=} (\exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)])$ separately.

- The effect of assignments in π on Q is characterised by $Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 1)]$. We obtain:

$$\begin{aligned} Q[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 1)] \;\equiv\; \exists j \in [0, n-1) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(f_{\mathtt{X}}(\mathtt{X}, 1), j)] \;\equiv \\ \exists j \in [1, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)] \end{aligned}$$

- Assumptions in π contribute the disjunct $wlp(\pi, \mathsf{F})$.

By combining both contributions into one term we obtain

$$wlp(\pi^n, \mathsf{F}) \equiv (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, 0)] \vee \exists j \in [1, n) \,.\, (wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]\,,$$

which establishes the claim of Lemma 1.

Accordingly, given a path π modifying the set of variables $\mathtt{X}$ and a corresponding set $f_{\mathtt{X}}$ of closed-form assignments, we can construct an accurate representation of π^n as follows:

$$\underbrace{[\,\forall j \in [0, n) \,.\, (\neg wlp(\pi, \mathsf{F}))\,[\mathtt{X}/f_{\mathtt{X}}(\mathtt{X}, j)]\,]}_{\text{satisfiable if } \pi^n \text{ is feasible}} \quad ; \quad \underbrace{\mathtt{X} := f_{\mathtt{X}}(\mathtt{X}, n)}_{\text{assignments of } \pi^n} \tag{7}$$

We emphasise that our construction (unlike many acceleration techniques) does *not* restrict the assumptions in π to a limited class of relations on integers.

The construction of the path (7), however, does require closed forms of all assignments in π. Since we do not construct closed forms for array assignments (as opposed to assignments to array indices, c.f. §3.2), we cannot apply Lemma 1 if $wlp(\pi, \mathsf{F})$ refers to an array assigned in π. In this case, we do not construct $\overset{\rightsquigarrow}{\pi}$.

For assignments of variables not occurring in $wlp(\pi, \mathsf{F})$, we augment the domain(s) of the variables $\mathtt{X}$ with an undefined value $\perp$ (implemented using a Boolean flag) and replace $f_{\mathtt{x}}$ with $\perp$ whenever the respective closed form is not available. Subsequently, whenever the search algorithm encounters an (abstract) counterexample, we use slicing to determine whether the feasibility of the counterexample depends on an undefined value $\perp$. If this is the case, the counterexample needs to be dismissed. Thus, any path $\overset{\rightsquigarrow}{\pi}$ containing references to $\perp$ is an *under-approximation* of π^n rather than an acceleration of π.

Example 3. For a path $\pi \overset{\text{def}}{=} [\mathtt{x} < 10]; \mathtt{x} := \mathtt{x} + 1; \mathtt{y} := \mathtt{y}^2$, we obtain the under-approximation $\overset{\rightsquigarrow}{\pi} \equiv n := *; [\forall j \in [0, n). \mathtt{x} + j < 10]; \mathtt{x} := \mathtt{x} + n; \mathtt{y} := \perp$. A counterexample traversing $\overset{\rightsquigarrow}{\pi}$ is feasible if its conditions do not depend on $\mathtt{y}$.

3.4 Arithmetic Overflows

The fact that the techniques in §3.1 used to derive closed forms do not take arithmetic overflow into account may lead to undesired effects. For instance, the assumption made in Example 2 that the characteristic function of the predicate $(\mathtt{i} + n < \mathtt{BUFLEN})$ is monotonic in n does not hold in the context of bit-vectors or modular arithmetic. Since, moreover, the behaviour of arithmetic over- or under-flow in C is not specified in certain cases, we conservatively rule out all occurrences thereof in $\overset{\rightsquigarrow}{\pi}$. For the simple assignment $\mathtt{i} := \mathtt{i} + n$ in Example 2, this can be achieved by adding the assumption $(\mathtt{i} + n \leq 2^l - 1)$ to $\overset{\rightsquigarrow}{\pi}$ (for unsigned l-bit vectors). In general, we have to add respective assumptions $(e_1 \otimes e_2 \leq 2^l - 1)$ for all arithmetic (sub-)expressions $e_1 \otimes e_2$ of bit-width l and operations $\otimes$ in $\overset{\rightsquigarrow}{\pi}$.

While this approach is *sound* (eliminating paths from $\overset{\rightsquigarrow}{\pi}$ does not affect the correctness of the instrumented program, since all behaviours following an overflow are still reachable via non-approximated paths), it imposes restrictions on the range of n. Therefore, the resulting approximation $\overset{\rightsquigarrow}{\pi}$ deviates from the acceleration π^* of π. Unlike acceleration over linear affine relations, this adjustment makes our approach bit-level accurate. We emphasise that the benefit of the instrumentation can still be substantial, since the number of iterations required to trigger an arithmetic overflow is typically large.

4 Eliminating Quantifiers from Approximations

A side effect of the approximation steps in §3.2 and §3.3 is the introduction of quantified assumptions. While quantification is often unavoidable in the presence of arrays, it is a detriment to performance of the decision procedures underlying the verification tools. In the worst case, quantifiers may result in the undecidability of path feasibility.

In the following, we discuss two techniques to eliminate or reduce the number of quantifiers in assumptions occurring in $\overset{\rightsquigarrow}{\pi}$.

4.1 Eliminating Quantifiers over Monotonic Predicates

We show that the quantifiers introduced by the technique presented in §3.3 can be eliminated if the predicate is monotonic in the quantified parameter.

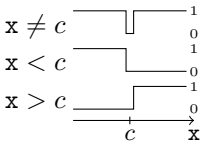
Definition 1 (Representing Function, Monotonicity). *The representing function f_P of a predicate P with the same domain takes, for each domain value, the value 0 if the predicate holds, and 1 if the predicate evaluates to false, i.e., $P(\mathtt{X}) \Leftrightarrow f_P(\mathtt{x}) = 0$. A predicate $P(n) : \mathbb{N} \to \mathbb{B}$ is monotonically increasing (decreasing) if its representing function $f_P(n) : \mathbb{N} \to \mathbb{N}$ is monotonically increasing (decreasing), i.e., $\forall m, n \,.\, m \leq n \Rightarrow f_P(m) \leq f_P(n)$.*

We extend this definition to predicates over variables $\mathtt{X}$ and $n \in \mathbb{N}$ as follows: $P(\mathtt{X}, n)$ is monotonically increasing in n if $(m \leq n) \wedge P(\mathtt{X}, n) \wedge \neg P(\mathtt{X}, m)$ is unsatisfiable.

Proposition 1. *$P(\mathtt{X}, n-1) \equiv \forall i \in [0, n) \,.\, P(\mathtt{X}, i)$ if P is monotonically increasing in i.*

The validity of Proposition 1 follows immediately from the definition of monotonicity. Accordingly, it is legitimate to replace universally quantified predicates in $\overset{\rightsquigarrow}{\pi}$ with their corresponding unquantified counterparts (c.f. Proposition 1).

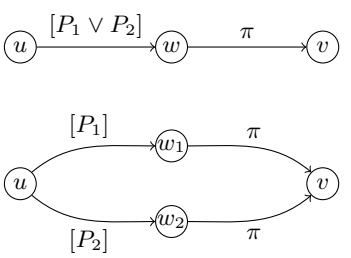
This technique, however, fails for simple cases such as $\mathtt{x} \neq c$ (c being a constant). In certain cases, the approach can still be applied after *splitting* a non-monotonic predicate P into monotonic predicates $\{P_1, \ldots, P_m\}$ such that $P \equiv \bigvee_{i=1}^{m} P_i$ (as illustrated in the Figure to the right). Subsequently, the path π guarded by P can be split as outlined in Figure 3. This transformation preserves reachability (a proof for $m = 2$ is given in Figure 3).



This approach is akin to trace partitioning [7], however, our intent is quantifier elimination rather than refining an abstract domain. We rely on a template-based approach to identify predicates that can be split (a constraint solver-based approach is bound to fail if c is symbolic). While this technique effectively deals with a broad number of standard cases, it does fail for quantifiers over array indices, since the array access operation is not monotonic.

4.2 Eliminating Quantifiers in Membership Tests for Array Indices

This sub-section aims at replacing the existentially quantified membership test in Predicate (6) by a quantifier-free predicate. To define a set of sufficient (but not necessary) conditions for when this is possible, we introduce the notion of increasing and dense array indices (c.f. [11]):



$$sp([P_1 \vee P_2]; \pi, Q) \equiv sp(\pi, (P_1 \wedge Q) \vee (P_2 \wedge Q)) \equiv sp(\pi, P_1 \wedge Q) \vee sp(\pi, P_2 \wedge Q)$$

$$sp(([P_1]; \pi) \square ([P_2]; \pi), Q) \equiv sp([P_1]; \pi, Q) \vee sp([P_2]; \pi, Q) \equiv sp(\pi, (P_1 \wedge Q)) \vee sp(\pi, (P_2 \wedge Q))$$

Fig. 3. Splitting disjunctive assumptions preserves program behaviour

Definition 2 (Increasing and Dense Variables). *A scalar variable* $\mathtt{x}$ *is (strictly)* increasing *in* π^n *iff* $\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1 \rangle} \geq \mathtt{x}^{\langle j \rangle}$ *(*$\forall j \in [0, n) \,.\, \mathtt{x}^{\langle j+1 \rangle} > \mathtt{x}^{\langle j \rangle}$*, respectively). Moreover, an increasing variable* $\mathtt{i}$ *is* dense *iff*

$$\forall j \in [0, n) \,.\, \left(\mathtt{x}^{\langle j+1 \rangle} = \mathtt{x}^{\langle j \rangle}\right) \vee \left(\mathtt{x}^{\langle j+1 \rangle} = \mathtt{x}^{\langle j \rangle} + 1\right) .$$

Variables decreasing in π^n *are defined analogously. A variable is monotonic (in* π^n*) if it is increasing or decreasing (in* π^n*).*

Note that if the closed form $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, n)$ of a variable $\mathtt{x}$ is a linear polynomial, then $\mathtt{x}$ is necessarily monotonic. The following proposition uses this property:

Proposition 2. *Let* $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *be the closed form (2) of* $\mathtt{x}^{\langle j \rangle}$*, where* $\alpha_{(2 \cdot k+2)} = 0$*, i.e., the polynomial* $f_{\mathtt{x}}$ *is linear. Then* $\Delta f_{\mathtt{x}} \stackrel{\text{def}}{=} f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j+1) - f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *(for* $j \in [0, n)$*) is the (symbolic) constant* $\sum_{i=1}^{k} \alpha_{(k+i)} \cdot \mathtt{x}_i^{\langle 0 \rangle} + \alpha_{(2 \cdot k+1)}$*. The variable* $\mathtt{x}$ *is (strictly) increasing in* π^n *if* $\Delta f_{\mathtt{x}} \geq 0$ *(*$\Delta f_{\mathtt{x}} > 0$*, respectively) and dense if* $0 \leq \Delta f_{\mathtt{x}} \leq 1$*.*

Lemma 2. *Let* $f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j)$ *be a linear polynomial representing the closed form (2) of* $\mathtt{x}^{\langle j \rangle}$ *(as in Proposition 2). The following logical equivalence holds:*

$$\exists j \in [0, n) \,.\, \mathtt{x} = f_{\mathtt{x}}(\mathtt{X}^{\langle 0 \rangle}, j) \;\equiv\; \begin{cases} ((\mathtt{x} - \mathtt{x}^{\langle 0 \rangle}) \mod \Delta f_{\mathtt{x}} = 0) \wedge (\frac{\mathtt{x} - \mathtt{x}^{\langle 0 \rangle}}{\Delta f_{\mathtt{x}}} < n) & \text{if } \mathtt{x} \text{ is strictly increasing} \\ \mathtt{x} - \mathtt{x}^{\langle 0 \rangle} \leq (n-1) \cdot \Delta f_{\mathtt{x}} & \text{if } \mathtt{x} \text{ is dense} \\ \mathtt{x} - \mathtt{x}^{\langle 0 \rangle} < n & \text{if both of the above hold} \end{cases} \tag{8}$$

The validity of Lemma 2 follows immediately from Proposition 2. Using Lemma 2, we can replace the existentially quantified membership test in Predicate (6) by a quantifier-free predicate if one of the side conditions in (8) holds. Given that the path prefix reaches the entry node of a loop, these conditions $\Delta f_{\mathtt{x}} > 0$ and $0 \leq \Delta f_{\mathtt{x}} \leq 1$ can be checked using a satisfiability solver.

Example 4. Let $\pi \stackrel{\text{def}}{=} \mathtt{a}[\mathtt{x}] := \mathtt{x};\ \mathtt{x} := \mathtt{x} + \mathtt{1}$ be the body of a loop. By instantiating (6), we obtain the condition

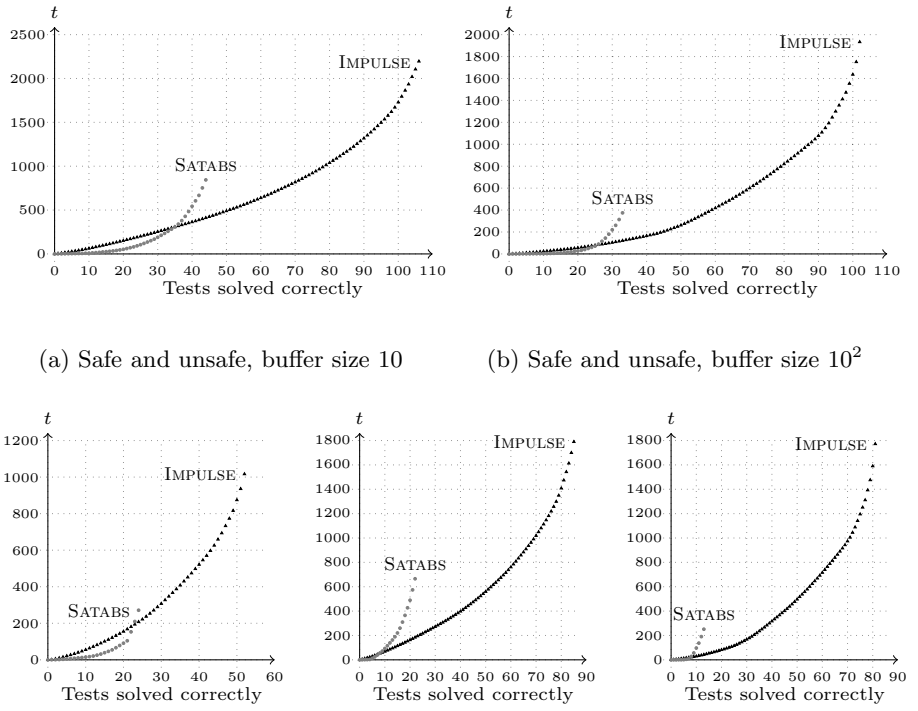
$$\forall j \in [0, n) \,.\mathtt{a}[\text{`}\mathtt{x} + j] = \text{`}\mathtt{x} + j \ \wedge \forall i. \left(\exists j \in [0, n) \,.\, i = \text{`}\mathtt{x} + j\right) \ \vee \ \left(\mathtt{a}[i] = \text{`}\mathtt{a}[i]\right),$$

in which the existentially quantified term can be replaced by $\mathtt{x} - \text{`}\mathtt{x} < n$.

5 Implementation and Experimental Results

Our under-approximation technique is designed to extend existing verifiers. To demonstrate its versatility, we implemented Impulse, a tool combining under-approximation with the two popular software verification techniques lazy abstraction with interpolants (LAwI) [16] and bounded model checking (specifically, Cbmc [4]). The underlying SMT solver used throughout was version 4.2 of Z3. Impulse comprises two phases:

1. Impulse first explores the paths of the CFG following the LAwI paradigm. If Impulse encounters a path containing a loop with body π, it computes $\overset{\rightsquigarrow}{\pi}$ (processing inner loops first in the presence of nested loops), augments the CFG accordingly, and proceeds to phase 2.
2. Cbmc inspects the instrumented CFG up to an iteration bound of 2. If no counterexample is found, Impulse returns to phase 1.



(a) Safe and unsafe, buffer size 10

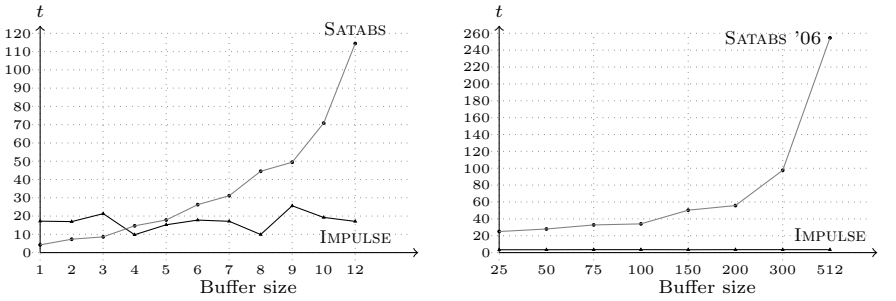
(b) Safe and unsafe, buffer size 10^2

(c) Safe/unsafe, b.-size 10^3

(d) Unsafe, buffer size 10

(e) Unsafe, buffer size 10^2

Fig. 4. Verification run-times (cumulative) of Verisec benchmark suite



(a) Single VERISEC test, varying buffer size (b) SATABS w. loop detect on Aeon 0.02a

Fig. 5. Run-time dependency on buffer size for unsafe benchmarks

In phase 1, spurious counterexamples serve as a catalyst to refine the current approximation of safely reachable states, relying on the weakest precondition[2] to generate the required Hoare triples. Phase 2 takes advantage of the aggressive path merging performed by CBMC, enabling fast counterexample detection.

We evaluated the effectiveness of under-approximation on the VERISEC benchmark suite [15], which consists of manually sliced versions of several open source programs that contain buffer overflow vulnerabilities. Of the 284 test cases of VERISEC, 144 are labelled as containing a buffer overflow, and 140 are labelled as safe.[3] The safety violations range from simple unchecked string copy into static buffers, up to complex loops with pointer arithmetic. The buffer size in each benchmark (c.f. `BUFLEN` in Figure 1) is adjustable and controls the depth of the counterexample. We compared our tool with SATABS (which outperforms IMPULSE w/o approximation[6]) on buffer sizes of 10, 100 and 1000, with a time limit of 300 s and a memory limit of 2 GB on an 8-core 3 GHz Xeon CPU. Figures 4a through 4c show the cumulative run-time for the whole benchmark suite, whereas Figures 4d and 4e show only unsafe program instances. Under-approximation did not improve (or impair) the run-time on safe instances.[4]

Figure 5 demonstrates that the time IMPULSE requires to detect a buffer overflow does not depend on the buffer size. Figure 5a compares SATABS and IMPULSE on a single VERISEC benchmark with a varying size parameter, showing that SATABS takes time exponential in the size of the buffer. Figure 5b provides a qualitative comparison of the loop-detection technique presented in [13] with IMPULSE on the Aeon 0.02a mail transfer agent. Figure 5b shows the run-times of SATABS'06 with loop detection as reported in [13],[5] as well as the run-times of IMPULSE on the same problem instances and buffer sizes. SATABS'06 outperforms

[2] In a preliminary interpolation-based implementation, Z3 was in many cases unable to provide interpolants for path formulas $\overrightarrow{\pi}$ with quantifiers, arrays, and bit-vectors.

[3] Our new technique discovered bugs in 10 of the benchmarks that had been labelled safe. SATABS timed out before identifying these bugs.

[4] The respective results are available on `http://www.cprover.org/impulse`.

[5] Unfortunately, loop detection in SATABS is neither available nor maintained anymore.

similar model checking tools that do not feature loop-handling mechanisms [13]. However, the run-time still increases exponentially with the size of the buffer, since the technique necessitates a validation of the unwound counterexample. IMPULSE does not require such a validation step.

6 Related Work

The under-approximation technique presented in this paper is based on our previous work on loop detection [13,14]. The algorithm in [13], however, does not yield a strict under-approximation, and thus necessitates an additional step to validate the unwound counterexample. Our new technique avoids this problem.

The techniques in §3.1 constitute a simple form of acceleration [3,6]. The subsequent restrictions in §3.3 and §3.4 on $\overrightarrow{\pi}$, however, impose a symbolic bound on the number of iterations, yielding an under-approximation. In contrast to acceleration of integer relations, our approximation is sound for bit-vector arithmetic. Sinha uses term rewriting to compute symbolic states parametrised by the loop counter, stating that his technique can be extended to support bit-vectors [19].

The quantifier elimination technique of §4.1 bears similarities with splitter predicates [18], a program transformation facilitating the generation of disjunctive invariants. Similarly, trace partitioning [7] splits program paths to increase the precision of static analyses. Neither technique aims at eliminating quantifiers.

Loop summarisation [12] and path invariants [2] avoid loop unrolling by selecting an appropriate over-approximation of the loop from a catalogue of invariant templates. Over-approximations are also used in the context of loop bound inference [20] and reasoning about termination [5]. However, over-approximations do not enable the efficient detection of counterexamples.

Hojjat et al. [9] uses interpolation to derive inductive invariants from accelerated paths. While this work combines under- and over-approximation, it is not aimed at counterexample detection. Motivated by the results of [9], we believe that under-approximation can, if combined with interpolation, improve the performance of verification tools on safe programs. We plan to support interpolation in a future version of our implementation.

We refer the reader to [14] for a description of additional related work.

7 Conclusion and Future Work

We present a sound under-approximation technique for loops in C programs with bit-vector semantics. The approach is very effective for finding deep counterexamples in programs that manipulate arrays, and compatible with a variety of existing verification techniques. A short-coming of our under-approximation technique is its lack of support for dynamic data structures, which we see as a challenging future direction.

References

1. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and Static Driver Verifier: Technology transfer of formal methods inside Microsoft. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 1–20. Springer, Heidelberg (2004)

2. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: PLDI, pp. 300–309. ACM (2007)
3. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces. PhD thesis, Université de Liège (1999)
4. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
5. Cook, B., Podelski, A., Rybalchenko, A.: Proving program termination. Commun. ACM 54(5), 88–98 (2011)
6. Finkel, A., Leroux, J.: How to compose presburger-accelerations: Applications to broadcast protocols. In: Agrawal, M., Seth, A.K. (eds.) FSTTCS 2002. LNCS, vol. 2556, pp. 145–156. Springer, Heidelberg (2002)
7. Handjieva, M., Tzolovski, S.: Refining static analyses by trace-based partitioning using control flow. In: Levi, G. (ed.) SAS 1998. LNCS, vol. 1503, pp. 200–214. Springer, Heidelberg (1998)
8. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70. ACM (2002)
9. Hojjat, H., Iosif, R., Konečný, F., Kuncak, V., Rümmer, P.: Accelerating interpolants. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 187–202. Springer, Heidelberg (2012)
10. Jhala, R., McMillan, K.L.: A practical and complete approach to predicate refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
11. Kovács, L., Voronkov, A.: Finding loop invariants for programs over arrays using a theorem prover. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 470–485. Springer, Heidelberg (2009)
12. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)
13. Kroening, D., Weissenbacher, G.: Counterexamples with loops for predicate abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
14. Kroening, D., Weissenbacher, G.: Verification and falsification of programs with loops using predicate abstraction. Formal Aspects of Computing 22, 105–128 (2010)
15. Ku, K., Hart, T.E., Chechik, M., Lie, D.: A buffer overflow benchmark for software model checkers. In: ASE, pp. 389–392. ACM (2007)
16. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
17. Nelson, G.: A generalization of Dijkstra's calculus. TOPLAS 11(4), 517–561 (1989)
18. Sharma, R., Dillig, I., Dillig, T., Aiken, A.: Simplifying loop invariant generation using splitter predicates. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 703–719. Springer, Heidelberg (2011)
19. Sinha, N.: Symbolic program analysis using term rewriting and generalization. In: Formal Methods in Computer-Aided Design, pp. 1–9 (2008)
20. Zuleger, F., Gulwani, S., Sinn, M., Veith, H.: Bound analysis of imperative programs with the size-change abstraction. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 280–297. Springer, Heidelberg (2011)

Proving Termination Starting from the End

Pierre Ganty[1,⋆] and Samir Genaim[2,⋆⋆]

[1] IMDEA Software Institute, Madrid, Spain
[2] Universidad Complutense de Madrid, Spain

Abstract. We present a novel technique for proving program termination which introduces a new dimension of modularity. Existing techniques use the program to incrementally construct a termination proof. While the proof keeps changing, the program remains the same. Our technique goes a step further. We show how to use the current partial proof to partition the transition relation into those behaviors known to be terminating from the current proof, and those whose status (terminating or not) is not known yet. This partition enables a new and unexplored dimension of incremental reasoning on the program side. In addition, we show that our approach naturally applies to conditional termination which searches for a precondition ensuring termination. We further report on a prototype implementation that advances the state-of-the-art on the grounds of termination and conditional termination.

1 Introduction

The question of whether or not a given program has an infinite execution is a fundamental theoretical question in computer science but also a highly interesting question for software practitioners. The first major result is that of Alan Turing, showing that the *termination problem* is undecidable. Mathematically, the termination problem for a given program *Prog* is equivalent to deciding whether the transition relation R induced by *Prog* is well-founded.

The starting point of our paper, is a result showing that the well-foundedness problem of a given relation R is equivalent to the problem of asking whether the transitive closure of R, noted R^+, is disjunctively well-founded [22]. That is whether R^+ is included in some W (in which case W is called a *transition invariant*) such that $W = W_1 \cup \cdots \cup W_n$, $n \in \mathbb{N}$ and each W_i is well-founded (in which case W is said to be *disjunctively well-founded*). This result has important practical consequences because it triggered the emergence of effective techniques, based on transition invariants, to solve the termination problem for real-world programs [11,2,27,18].

By replacing the well-foundedness problem of R with the equivalent disjunctive well-foundedness problem of R^+, one allows for the incremental construction of W: when the inclusion of R^+ into W fails then use the information from the failure to update W with a further well-founded relation [10]. Although the proof is incremental for W, it is

⋆ Supported by the Spanish projects with references TIN2010-20639, TIN2012-39391-C04 and the Danish project with grant number 10-084290.

⋆⋆ Supported by the EU project ICT-231620 *HATS*, and the Spanish projects TIN2008-05624 *DOVES*, S2009TIC-1465 *PROMETIDOS-CM* and TIN2012-38137 *VIVAC*.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 397–412, 2013.

important to note that a similar result does not hold for R. That is, it is in general not true that given $R = R_1 \cup R_2$, if $R_1^+ \subseteq W$ and $R_2^+ \subseteq W$ then $R^+ \subseteq W$.

We introduce a new technique that, besides being incremental for W, further partitions the transition relation R separating those behaviors known to be terminating from the current W, from those whose status (terminating or not) is not known yet. Formally, given R and a candidate W, we shall see how to compute a partition $\{R_G, R_B\}$ of R such that (*a*) $R_G^+ \subseteq W$; and (*b*) every infinite sequence $s_1\ R\ s_2\ R \cdots s_i\ R\ s_{i+1} \cdots$ (or *trace*) has a suffix that exclusively consists of transitions from R_B, namely we have $s_z\ R_B\ s_{z+1}\ R_B \cdots$ for some $z \geq 1$.

It follows that well-foundedness of R_B implies that of R. Consequently, we can focus our effort exclusively on proving well-foundedness of R_B. In the affirmative, then so is R and hence termination is proven. In the negative, then we have found an infinite trace in R_B, hence in R. We observed that working with R_B typically provides further hints on which well-founded relations to add to W. The partition of R into $\{R_G, R_B\}$ enables a new and unexplored dimension of modularity for termination proofs.

Let us mention that the partitioning of R is the result of adopting a fixpoint centric view on the disjunctive well-foundedness problem and leverage equivalent formulation of the inclusion check. More precisely, we introduce the dual of the check $R^+ \subseteq W$ by defining the adjoint to the function $\lambda X.\, X \circ R$ used to define R^+. Without defining it now, we write the dual check as follows: $R \subseteq W^-$. We shall see that while the failure of $R^+ \subseteq W$ provides information to update W; the failure of $R \subseteq W^-$ provides information on *all pairs* in R responsible for the failure of W as a transition invariant. This is exactly that information, of semantical rather than syntactical nature, that we use to partition R.

We show that the partitioning of R can be used not only for termination, but it also serves for conditional termination. The goal here is to compute a precondition, that is a set $\mathcal{P}$ of states, such that no infinite trace starts from a state of $\mathcal{P}$. We show how to compute a (non-trivial) precondition from the relation R_B.

Our contributions are summarized as follows: (*i*) we present `Acabar`, a new algorithm which allows for enhanced modular reasoning about infinite behaviors of programs; (*ii*) we show that, besides termination, `Acabar` can be used in the context of conditional termination; and (*iii*) finally, we report on a prototype implementation of our techniques and compare it with the state-of-the-art on two grounds: the termination problem, and the problem of inferring a precondition that guarantees termination.

2 Example

In this section, we informally overview our proposed techniques on an example taken from the literature [9]. Consider the following loop:

```
while ( x>0 ){ x:=x+y; y:=y+z; }
```

represented by the transition relation $R = \{x > 0, x' = x + y, y' = y + z, z' = z\}$, where the primed variables represent the values of the program variables after executing the loop body. Note that, depending on the input values, the program may not terminate (e.g. for $x = 1$, $y = 1$ and $z = 1$). Below we apply `Acabar` to prove termination. As we will see, this attempt ends with a failure which provide information on which subset

of the transition relation to blame. Then, we will explain how to compute a termination precondition from this subset.

In order to prove termination of this loop, we seek a disjunctive well-founded relation W such that $R^+ \subseteq W$. To find such a W, `Acabar` is supported by incrementally (and automatically) inferring (potential) linear ranking functions for R or R^+ [9,10]. When running on R, `Acabar` first adds the candidate well-founded relation $W_1 = \{x' < x, x > 0\}$ to W which is initially empty. Relation W_1 stems from the observation that, in R, x is bounded from below (as shown by the guard) but not necessarily decreasing. Hence, using $W = W_1$, `Acabar` partitions R into $\{R_G^{(1)}, R_B^{(1)}\}$ where:

$$\begin{aligned} R_G^{(1)} &= \{x > 0, x' = x + y, y' = y + z, z' = z, y < 0, z \leq 0\} \\ R_B^{(1)} &= \{x > 0, x' = x + y, y' = y + z, z' = z, y < 0, z > 0\} \vee \\ &\quad \{x > 0, x' = x + y, y' = y + z, z' = z, y \geq 0\} \ . \end{aligned}$$

The partition comes with the further guarantee that *every infinite trace in R must have a suffix that exclusively consists of transitions from $R_B^{(1)}$*, which means that if $R_B^{(1)}$ is well-founded then so is R. In addition, one can easily see that $(R_G^{(1)})^+ \subseteq W$.

Next, `Acabar` calls itself recursively on $R_B^{(1)}$ to show its well-foundedness. As before, it first adds $W_2 = \{y' < y, y \geq 0\}$ to W. Similarly to the construction of W_1, W_2 stems from the observation that, in some parts of $R_B^{(1)}$, y is bounded from below but not necessarily decreasing. Then, using $W = W_1 \vee W_2$, `Acabar` partitions $R_B^{(1)}$ into:

$$\begin{aligned} R_G^{(2)} &= \{x > 0, x' = x + y, y' = y + z, z' = z, z < 0\} \\ R_B^{(2)} &= \{x > 0, x' = x + y, y' = y + z, z' = z, y \geq 0, z \geq 0\} \ . \end{aligned}$$

Again the partition $\{R_G^{(2)}, R_B^{(2)}\}$ of $R_B^{(1)}$ comes with a similar guarantee. This time it holds that that every infinite trace in R must have a suffix that exclusively consists of transitions from $R_B^{(2)}$. Recursively applying `Acabar` on $R_B^{(2)}$ does not yield any further partitioning, that is $R_B^{(3)} = R_B^{(2)}$. The reason being that no potential ranking function is automatically inferred. Thus, `Acabar` fails to prove well-foundedness of R, which is indeed not well-founded. However, due to the above guarantee, we can use $R_B^{(2)}$ to infer a sufficient precondition for the termination of R. We explain this next.

Inferring a sufficient precondition is done in two steps: (*i*) we infer (an overapproximation of) the set of all states $\mathcal{Z}$ visited by some infinite sequence of steps in $R_B^{(2)}$; and (*ii*) we infer (an overapproximation of) the set of all states $\mathcal{V}$ each of which can reach $\mathcal{Z}$ through some steps in R. Turning to the example, we infer $\mathcal{Z} = \{x > 0, y \geq 0, z \geq 0\}$ and the following overapproximation $\mathcal{V}'$ of $\mathcal{V}$:

$$\mathcal{V}' = \{x \geq 1, z = 0, y \geq 0\} \vee \{x \geq 1, z \geq 1, x + y \geq 1, x + 2y + z \geq 1, x + 3y + 3z \geq 1\} \ .$$

It can be seen that every infinite trace visits only states in $\mathcal{V}'$, hence the complement of $\mathcal{V}'$ is a precondition for termination.

Let us conclude this section by commenting on an example for which `Acabar` proves termination. Assume that we append `z:=z-1` to the loop body above and call R' the induced transition relation. Following our previous explanations, running `Acabar` on R' updates W from $\emptyset$ to W_1, and then to $W_1 \vee W_2$. Then, and contrary to the previous explanations, `Acabar` will further update W to $W_1 \vee W_2 \vee W_3$ where W_3 is the well-founded relation $\{z' < z, z \geq 0\}$. From there, `Acabar` returns with value $R_B^{(3)} = \emptyset$, hence we have that R' is well-founded.

3 Preliminaries

A *transition system* is a pair (Q, R) where Q is the set of *states* and $R \subseteq Q \times Q$ is the *transition relation*. An *initialized* transition system includes a further component $\mathcal{I} \subseteq Q$, the set of *initial states*. For simplicity, we defer the treatment of initial states to Sec. 8.

An *R-trace* is a sequence $s_1, s_2, \ldots, s_n$ of states such that for every i, $1 \leq i < n$ we have $(s_i, s_{i+1}) \in R$. When R is clear from the context we simply say trace. An *infinite R-trace* is a sequence $s_1, s_2, \ldots$ of states such that for every $i \geq 1$ we have $(s_i, s_{i+1}) \in R$. Given $R' \subseteq R$ and an infinite R-trace π we say that π has *infinitely many steps* in R' if $(s_i, s_{i+1}) \in R'$ for infinitely many $i \geq 1$.

Given a relation $R' \subseteq R$ and a set $Q' \subseteq Q$, define $post[R'](Q') \stackrel{\text{def}}{=} \{s' \in Q \mid \exists s \in Q' : (s, s') \in R'\}$. We say that this operator computes the *R'-successors of Q'*. Dually, define $pre[R'](Q') \stackrel{\text{def}}{=} post[R'^{-1}](Q') = \{s \in Q \mid \exists s' \in Q' : (s, s') \in R'\}$. We say that this operator computes the *R'-predecessors of Q'*.[1]

A relation $W \subseteq Q \times Q$ is called *disjunctively well-founded* iff W coincides with the union of finitely many relations (viz. $W = W_1 \cup \ldots \cup W_n$) each of which is well-founded (viz. there is no infinite sequence $s_1, s_2, \ldots$ such that $(s_i, s_{i+1}) \in W_\ell$ for all $i \geq 1$).

In this paper, we adhere to the following conventions: calligraphic letters $\mathcal{X}, \mathcal{Y}, \ldots$ refer to subsets of Q and capital letters $X, Y, \ldots$ refer to relations over Q, that is subsets of $Q \times Q$. Further, throughout the paper the letter W is used to denote a relation over Q that is disjunctively well-founded.

A *linear expression* is of the form $a_0 + a_1 x_1 + \cdots + a_n x_n$ where $a_i \in \mathbb{Z}$ and $\bar{x} = \langle x_1, \ldots, x_n \rangle$ are *variables* ranging over $\mathbb{Z}$. An *atomic linear constraint* c is of the form $e_1 \; op \; e_2$ where e_i is a linear expression and $op \in \{=, \geq, \leq, >, <\}$. A *formula* ψ is a Boolean combination of atomic linear constraints. Note that $\neg\psi$ is also a formula. For the sake of simplicity, a conjunction $c_1 \wedge \cdots \wedge c_n$ of atomic linear constraints is sometimes written as the set $\{c_1, \ldots, c_n\}$. A *solution* of a formula ψ is a mapping from its variables into the integers such that the formula evaluates to true. Sets and relations over, respectively, $\mathbb{Z}^n$ and $\mathbb{Z}^n \times \mathbb{Z}^n$ are sometimes specified using formulas, with the customary convention, for relations, of variables and primed variables. For instance, the formula $\{x \geq 0, x' = x - y, y' = y\}$ defines the relation $R \subseteq \mathbb{Z}^2 \times \mathbb{Z}^2$ such that $R = \{\langle (x, y), (x', y') \rangle \mid x \geq 0 \wedge x' = x - y \wedge y' = y\}$.

Finally, we briefly recall classical results of lattice theory and refer to the classical book of Davey and Priestley [15] for further information. Let f be a function over a partially ordered set $(L, \sqsubseteq)$. A *fixpoint* of f is an element $l \in L$ such that $f(l) = l$. We denote by *lfp* f and *gfp* f, respectively, the *least* and the *greatest fixpoint*, when they exist, of f. The well-known Knaster-Tarski's theorem states that each order-preserving function $f \in L \to L$ over a complete lattice $\langle L, \sqsubseteq, \bigsqcup, \bigsqcap, \top, \bot \rangle$ admits a least (greatest) fixpoint and the following characterization holds:

$$lfp\ f = \bigsqcap \{x \in L \mid f(x) \sqsubseteq x\} \qquad gfp\ f = \bigsqcup \{x \in L \mid x \sqsubseteq f(x)\} \ . \tag{1}$$

[1] We define R^{-1}, R^* and R^+ to be $R^{-1} = \{(s', s) \mid (s, s') \in R\}$, $R^* = \bigcup_{i \geq 0} R^i$ and $R^+ = R \circ R^*$ where R^0 is the identity, $R^{i+1} = R^i \circ R$ and $R_1 \circ R_2 = \{(s, s'') \mid \exists s' : (s, s') \in R_1 \wedge (s', s'') \in R_2\}$.

4 Modular Reasoning for Termination

A termination proof based on transition invariants consists in establishing the existence of a disjunctively well-founded transition invariant. That is, the goal is to prove the inclusion of R^+, into some W.[2] For short, we write $R^+ \subseteq W$. Proving termination is thus reduced to finding some W and prove that the inclusion hold.

In the above inclusion check, R^+ coincides with the least fixpoint of the function $\lambda Y.\, R \cup g(Y)$ where $g \stackrel{\text{def}}{=} \lambda Y.\, Y \circ R$. It is known [13] that if we can find an *adjoint function* $\tilde{g}$ to g such that $g(X) \subseteq Y$ iff $X \subseteq \tilde{g}(Y)$ for all X, Y then there exists an equivalent inclusion check to $R^+ \subseteq W$. This equivalent check, denoted $R \subseteq W^-$ in the introduction, is such that W^- is defined as a greatest fixpoint of the function $\lambda Y.\, W \cap \tilde{g}(Y)$. Next, we define $\tilde{g} \stackrel{\text{def}}{=} \lambda Y.\, \neg(\neg Y \circ R^{-1})$.

Lemma 1. *Let X, Y be subsets of $Q \times Q$ we have: $X \circ R \subseteq Y \Leftrightarrow X \subseteq \neg(\neg Y \circ R^{-1})$.*

Proof. First we need an easily proved logical equivalence:

$$(\varphi_1 \wedge \varphi_2) \Rightarrow \varphi_3 \text{ iff } (\neg\varphi_3 \wedge \varphi_2) \Rightarrow \neg\varphi_1 \ .$$

Then we have:

$$\begin{aligned}
& X \circ R \subseteq Y \\
\text{iff } & \forall s, s', s_1 : ((s, s_1) \in X \wedge (s_1, s') \in R) \Rightarrow (s, s') \in Y \\
\text{iff } & \forall s, s', s_1 : ((s, s') \notin Y \wedge (s_1, s') \in R) \Rightarrow (s, s_1) \notin X && \text{by above equivalence} \\
\text{iff } & \forall s, s', s_1 : ((s, s') \notin Y \wedge (s', s_1) \in R^{-1}) \Rightarrow (s, s_1) \notin X && \text{def. of } R^{-1} \\
\text{iff } & \forall s, s', s_1 : ((s, s') \in \neg Y \wedge (s', s_1) \in R^{-1}) \Rightarrow (s, s_1) \in \neg X \\
\text{iff } & (\neg Y \circ R^{-1}) \subseteq \neg X \\
\text{iff } & X \subseteq \neg(\neg Y \circ R^{-1})
\end{aligned}$$

□

Intuitively, g corresponds to *forward reasoning* for proving termination while $\tilde{g}$ corresponds to *backward reasoning* because of the composition with R^{-1}. The least fixpoint $lfp\ \lambda Y.\, R \cup g(Y)$ is the least relation Z containing R and closed by composition with R, viz. $R \subseteq Z$ and $Z \circ R \subseteq Z$. On the other hand, the greatest fixpoint $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ is best understood as the result of removing from W all those pairs (s, s') of states such that $(s, s') \circ R^+ \not\subseteq W$. This process returns the largest subset Z' of W which is closed by composition with R, viz. $Z' \subseteq W$ and $Z' \circ R \subseteq Z'$. Using the results of Cousot [13] we find next that termination can be shown by proving either inclusion of Lem. 2.

Lemma 2 (from [13]). *$lfp\ \lambda Y.\, R \cup g(Y) \subseteq W \Leftrightarrow R \subseteq gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$.*

Proof.

$$\begin{aligned}
lfp\ \lambda Y.\, R \cup g(Y) \subseteq W \text{ iff } & \exists A : R \subseteq A \wedge g(A) \subseteq A \wedge A \subseteq W && \text{by (1)} \\
\text{iff } & \exists A : R \subseteq A \wedge A \subseteq \tilde{g}(A) \wedge A \subseteq W && \text{Lem. 1} \\
\text{iff } & R \subseteq gfp\ \lambda Y.\, W \cap \tilde{g}(Y) && \text{by (1)}
\end{aligned}$$

□

[2] Recall that W is always assumed to be disjunctively well-founded.

As we shall see, the inclusion check based on the greatest fixpoint has interesting consequences when trying to prove termination.

An important feature when proving termination using transition invariants is to define actions to take when the inclusion check $lfp\ \lambda Y.\, R \cup g(Y) \subseteq W$ fails. In this case, some information is extracted from the failure (e.g., a counter example), and is used to enrich W with more well-founded relations [10].

We shall see that, for the backward approach, failure of $R \subseteq gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ induces a partition of the transition relation R into $\{R_G, R_B\}$ such that (a) $(R_G)^+ \subseteq W$; together with the following termination guarantee (b) every infinite R-trace contains a suffix that is an infinite R_B-trace (Lem. 4). An important consequence of this is that we can focus our effort exclusively on proving termination of R_B. It is important to note that the guarantee that no infinite R-trace contains infinitely many steps from R_G is not true for any partition $\{R_G, R_B\}$ of R but it is true for our partition which we define next.

Definition 1. *Let $G = gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$, we define $\{R_G, R_B\}$ to be the partition of R given by $R_G = R \cap G$ and $R_B = R \setminus R_G$.*

Example 1. Let $R = \{x \geq 1, x' = x + y, y' = y - 1\}$ and assume $W = \{x' < x, x \geq 1\}$ which is well-founded, hence disjunctively well-founded as well. Evaluating the greatest fixpoint (we omit calculations) yields

$$R_G = \{x \geq 1, x' = x + y, y' = y - 1, y < 0\}$$
$$R_B = \{x \geq 1, x' = x + y, y' = y - 1, y \geq 0\}$$

which is clearly a partition of R. The relation R_G consists of those pairs of states where y is negative, hence x is decreasing as captured by W. On the other hand, R_B consists of those pairs where y is positive or null. It follows that, when taking a step from R_B, x does not decrease. This is precisely for those pairs that W fails to show termination. ■

Next, we state and prove the termination guarantees of the partition $\{R_G, R_B\}$.

Lemma 3. *Given R_G as in Def. 1 we have $lfp\ \lambda Y.\, R_G \cup Y \circ R \subseteq W$.*

Proof.

$$\begin{aligned}
& G \subseteq \tilde{g}(G) \wedge G \subseteq W && \text{def. of } G \text{ and (1)} \\
\text{only if } & g(G) \subseteq G \wedge G \subseteq W && \text{Lem. 1} \\
\text{only if } & R \cap G \subseteq G \wedge g(G) \subseteq G \wedge G \subseteq W && \\
\text{only if } & R_G \subseteq G \wedge g(G) \subseteq G \wedge G \subseteq W && \text{def. of } R_G \\
\text{only if } & lfp\ \lambda Y.\, R_G \cup g(Y) \subseteq W && \text{by (1)}
\end{aligned}$$

□

An equivalent formulation of the previous result is $R_G \circ R^* \subseteq W$, which in turn implies, since $R_G \subseteq R$, that $(R_G \circ R^*)^+ \subseteq W$, and also $(R_G)^+ \subseteq W$.

Lemma 4. *Every infinite R-trace has a suffix that is an infinite R_B-trace.*

Proof. Assume the contrary, i.e., there exists an infinite R-trace $s_1, s_2, \ldots$ that contains infinitely many steps from R_G. Let $S = s_{i_1}, s_{i_2}, \ldots$ be the infinite subsequence of states such that $(s_{i_j}, s_{i_j+1}) \in R_G$ for all $j \geq 1$. Recall also that $W = W_1 \cup \cdots \cup W_n$ where each W_ℓ is well-founded. For any $s_i, s_j \in S$ with $i < j$ it holds that $(s_i, s_j) \in R_G \circ R^*$, and thus, according to Lem. 3, we also have that $(s_i, s_j) \in W_\ell$ for some $1 \leq \ell \leq n$. Ramsey's theorem [24] guarantees the existence of an infinite subsequence $S' = s_{j_1}, s_{j_2}, \ldots$ of S, and a single W_ℓ, such that for all $s_i, s_j \in S'$ with $i < j$ we have $(s_i, s_j) \in W_\ell$. This contradicts that W_ℓ is well-founded and we are done. □

Remark 1. When fixpoints are not computable, they can be approximated from above or from below [14]. It is routine to check that the results of Lemmas 3 and 4 remain valid when replacing $G = gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ in Def. 1 with $G' \subseteq gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$. Therefore we have that, even when approximating $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ from below, the termination guarantees of $\{R_G, R_B\}$ still hold. In Sec. 6, we shall see how to exploit this result in practice.

Example 2 (cont'd from Ex. 1). We left Ex. 1 with $W = \{x' < x, x \geq 1\}$ and $R_B = \{x \geq 1, x' = x + y, y' = y - 1, y \geq 0\}$. As argued previously, to prove the well-foundedness of R it is enough to show that R_B is well-founded. For clarity, we rename R_B into $R_B^{(1)}$. Next we partition $R_B^{(1)}$ as we did it for R in Ex. 1. As a result, we update W by adding the well-founded relation $\{y' < y, y \geq 0\}$. Then we evaluate again G (we omit calculations) which yields $R_B^{(2)} = \emptyset$. Hence we conclude from Lem. 4 that R is well-founded. ■

Building upon all the previous results, we introduce `Acabar` that is given at Alg. 1. `Acabar` is a recursive procedure that takes as input two parameters: a transition relation R and a disjunctively well-founded relation W. The second parameter is intended for recursive calls, hence the user should invoke `Acabar` as follows: `Acabar`$(R, \emptyset)$. We call it the *root call*. Upon termination, `Acabar` returns a subset R_B of the transition relation R. If it returns the empty set, then the relation R is well-founded, hence termination is proven. Otherwise ($R_B \neq \emptyset$), we can not know for sure if R is well-founded: there might be an infinite R-trace. However, Lem. 4 tells us that every infinite R-trace must have a suffix that is an infinite R_B-trace. It may also be the case that R_B is well-founded (and so is R) in which case it was not discovered by `Acabar`. Another case is that $R = R_B$. In this case we have made no progress and therefore we stop. Whenever $R_B \neq \emptyset$, we call this returned value the *problematic* subset of R.

Next we study progress properties of `Acabar`. We start by defining the sequence $\{R^{(i)}\}_{i \geq 0}$ where each $R^{(i)}$ is the argument passed to the i-th recursive call to `Acabar`. In particular, $R^{(0)}$ is the argument of the root call. Furthermore, we define the sequences $\{R_B^{(i)}\}_{i \geq 1}$ and $\{R_G^{(i)}\}_{i \geq 1}$ where $\{R_G^{(i)}, R_B^{(i)}\}$ is a partition of $R^{(i-1)}$ and $R_B^{(i)} = R^{(i)}$ for all $i \geq 1$.

Lemma 5. *Given a run of* `Acabar` *with at least $i \geq 1$ recursive calls, then we have*

$$R^{(0)} \supsetneq R^{(1)} \supsetneq \cdots \supsetneq R^{(i)} \ .$$

Proof. The proof is by induction on i, for $i = 1$ it follows from the definitions that $R^{(1)} = R_B^{(1)}$ and $\{R_B^{(1)}, R_G^{(1)}\}$ is a partition of $R^{(0)}$. Moreover, since at least $i = 1$ recursive calls take place we find that the condition of line 5 fails, meaning neither $R_B^{(1)}$ nor $R_G^{(1)}$ is empty, hence $R^{(1)}$ is a strict subset of $R^{(0)}$. The inductive case is similar. □

Algorithm 1. Enhanced modular reasoning

```
Acabar(R,W)
Input: a relation R ⊆ Q × Q
Input: a relation W ⊆ Q × Q such that W is disjunctively well-founded
Output: R_B ⊆ R
1 begin
2     W := W∪find_dwf_candidate(R)
3     let G be such that G ⊆ gfp λY. W ∩ g̃(Y)
4     R_B := R \ G
5     if R_B = ∅ or R_B = R then
6         return R_B
7     else
8         return Acabar(R_B, W)
```

By Lemmas 4 and 5, we have that every infinite $R^{(0)}$-trace has a suffix that is an infinite $R_B^{(i)}$-trace for every $i \geq 1$. As a consequence, forcing `Acabar` to execute line 6 after predefined number of recursive calls, it returns a relation $R_B^{(i)}$ such that the previous property holds. Incidentally, we find that `Acabar` proves program termination when it returns the empty set as stated next.

Theorem 1. *Upon termination of the call* `Acabar`$(R, \emptyset)$*, if it returns the empty set, then the relation R is well-founded.*

Let us turn to line 2. There, `Acabar` calls a subroutine `find_dwf_candidate`(R) implementing a heuristic search which returns a disjunctively well-founded relation using hints from the representation and the domain of R. Details about its implementation, that is inspired from previous work [9,10], will be given at Sec. 6 — we will consider the case of R being a relation over the integers of the form $R = \rho_1 \vee \cdots \vee \rho_n$ where each ρ_i is a conjunction of linear constraints over the variables $\vec{x}$ and $\vec{x}'$. Let us intuitvely explain this procedure on an example.

Example 3 (cont'd from Ex. 2). `Acabar`$(R, \emptyset)$ updates W as follows: *(1)* $\emptyset$; *(2)* $\{x' < x, x \geq 1\}$; *(3)* $\{x' < x, x \geq 1\}, \{y' < y, y \geq 0\}$. The first update from $\emptyset$ to $\{x' < x, x \geq 1\}$ is the result of calling `find_dwf_candidate`(R). The hint used by `find_dwf_candidate` is that x is bounded from below in R. The second update to W results from calling `find_dwf_candidate`$(R_B = \{x \geq 1, x' = x + y, y' = y - 1, y \geq 0\})$. Since R_B has the linear ranking function $f(x, y) = y$, `find_dwf_candidate` returns $\{y' < y, y \geq 0\}$. ■

5 `Acabar` for Conditional Termination

As mentioned previously, upon termination, `Acabar` returns a subset R_B of the transition relation R. If this set is empty then R is well-founded and we are done. Otherwise, R_B is a non-empty subset and called the problematic set. In this section, we shall see how to compute, given the problematic set, a *precondition* $\mathcal{P}$ *for termination*. More precisely, $\mathcal{P}$ is a set of states such that no infinite R-trace starts with a state of $\mathcal{P}$. We illustrate our definitions using the simple but challenging example of Sec. 2.

Example 4. Consider again the relation $R = \{x > 0, x' = x + y, y' = y + z, z' = z\}$. Upon termination `Acabar` returns the following relation:

$$R_B = \{x' = x + y, y' = y + z, z' = z, x > 0, y \geq 0, z \geq 0\}$$

which corresponds to all the cases where x is stable or increasing over time. ■

Lemma 4 tells us that every infinite R-trace π is such that $\pi = \pi_f \pi_\infty$ where π_f is a finite R-trace and π_∞ is an infinite R_B-trace. Our computation of a precondition for termination is divided into the following parts: (*i*) compute those states $\mathcal{Z}$ visited by infinite R_B-trace; (*ii*) compute the set $\mathcal{V}$ of R^*-predecessors of $\mathcal{Z}$, that is the set of states visited by some R-trace ending in $\mathcal{Z}$; and (*iii*) compute $\mathcal{P}$ as the complement of $\mathcal{V}$. Formally, (*i*) is given by a greatest fixpoint expression $gfp\ \lambda\mathcal{X}.\, pre[R_B](\mathcal{X})$. This expression is directly inspired by the work of Bozga et al. [6] on deciding conditional termination. This greatest fixpoint is the largest set $\mathcal{Z}$ of states each of which has an R_B-successor in $\mathcal{Z}$. Because of this property, every infinite R_B-trace visits only states in $\mathcal{Z}$. In $\pi = \pi_f \pi_\infty$, this corresponds to the suffix π_∞ that is an infinite R_B-trace.

Example 5. For R_B as given in Ex. 4, we have that $\mathcal{Z} = \{z \geq 0, y \geq 0, x > 0\}$ which contains the following infinite R_B-trace:

$$(x = 1, y = 0, z = 0)\, R_B\, (x = 1, y = 0, z = 0)\, R_B\, (x = 1, y = 0, z = 0)\, R_B \ldots \quad ■$$

Let us now turn to (*ii*), that is computing the set $\mathcal{V}$ of R^*-predecessors of $\mathcal{Z}$. It is known that $\mathcal{V}$ coincides with $lfp\ \lambda\mathcal{X}.\, \mathcal{Z} \cup pre[R](\mathcal{X})$. Intuitively, we prepend to those infinite R_B-traces a finite R-trace. That is, prefixing π_f to π_∞ results in $\pi = \pi_f \pi_\infty$. Finally, step (*iii*) results into a precondition for termination $\mathcal{P}$ obtained by complementing $\mathcal{V}$.

Example 6. Computing $lfp\ \lambda\mathcal{X}.\, \mathcal{Z} \cup pre[R](\mathcal{X})$ for $\mathcal{Z}$ as given in Ex. 5 and $R = \{x' = x + y, y' = y + z, z' = z, x > 0, y \geq 0, z \geq 0\}$ (Ex. 4) gives $\mathcal{V} = \mathcal{V}_1 \vee \mathcal{V}_2$ where

$$\mathcal{V}_1 = \{x \geq 1, z = 0, y \geq 0\}$$
$$\mathcal{V}_2 = \{x \geq 1, z \geq 1\} \cup \{x + i * y + j * z \geq 1 \mid i \geq 1, j = \textstyle\sum_{k=0}^{i-1} k\}\ .$$

Intuitively, the set $\mathcal{V}_1$ of states corresponds to entering the loop with $z = 0$ and y non-negative, in which case the loop clearly does not terminate. The set $\mathcal{V}_2$ of states corresponds to entering the loop with z positive, and the loop does not terminate after i-th iterations for all i. Note that $\mathcal{V}_2$ consists of infinitely many atomic formulas. Complementing $\mathcal{V}$ gives $\mathcal{P}$. ■

Theorem 2. *There exists an infinite R-trace starting from s iff* $s \notin \mathcal{P}$.

Approximations. As argued previously, it is often the case that only approximations of fixpoints are available. In our case, any overapproximation of either $\mathcal{Z}$ or $\mathcal{V}$ can be exploited to infer $\mathcal{P}$. Because of approximations, we lose the if direction of the theorem, that is, we can only say that there is no infinite R-trace starting from some $s \in \mathcal{P}$.

Example 7. Using finite disjunctions of linear constraints, we can approximate $\mathcal{V}$ by

$$\{x \geq 1, z = 0, y \geq 0\} \vee \{x \geq 1, z \geq 1, x + y \geq 1, x + 2y + z \geq 1, x + 3y + 3z \geq 1\}$$

and then the complement $\mathcal{P}$ is

$$x \leq 0 \vee x + y \geq 1 \vee x + 2y + z \leq 0 \vee x + 3y + 3z \leq 0 \vee z \leq -1 \vee (y \leq -1 \wedge z \leq 0)$$

which is a sufficient precondition for termination. Note that the first 4 disjuncts correspond to the executions which terminates after 0, 1, 2 and 3 iterations. ■

6 Implementation

We have implemented the techniques described in Sec. 4 and 5 for the case of multiple-path integer linear-constraint loops. These loops correspond to relations of the form $R = \rho_1 \vee \cdots \vee \rho_d$ where each ρ_i is a conjunction of linear constraints over the variables $\bar{x}$ and $\bar{x}'$. In this context, the set Q of states is equal to $\mathbb{Z}^n$ where n is the number of variables in $\bar{x}$. This is a classical setting for termination [5,7,22]. Internally, we represent sets of states and relations over them as DNF formulas where the atoms are linear constraints. In what follows, we explain sufficient implementation details so that our experiments can be independently reproduced if desired. Our implementation is available [1].

We start with line 2 of Alg. 1. Recall that the purpose of this line is to add more well-founded relations to W based on the current relation R. In our implementation, W consists of well-founded relations of the form $\{f(\bar{x}) \geq 0, f(\bar{x}') < f(\bar{x})\}$ where f is a linear function [10,9]. Thus, our implementation looks for such well-founded relations. In particular, for each ρ_i of R we add new well-founded relations to W as follows: if ρ_i has a linear ranking function $f(\bar{x})$ that is synthesized automatically [21,5] then $\{f(\bar{x}') < f(\bar{x}), f(\bar{x}) \geq 0\}$ is added to W; otherwise, let $\{f_1(\bar{x}) \geq 0, \ldots, f_d(\bar{x}) \geq 0\}$ be the result of projecting each ρ_i on $\bar{x}$ (i.e., eliminating variables $\bar{x}'$ from ρ_i), then $\{\{f_i(\bar{x}') < f_i(\bar{x}), f_i(\bar{x}) \geq 0\} \mid 1 \leq i \leq d\}$ is added to W. Because f_i is bounded but not necessarily decreasing, it is called a *potential linear ranking function* [9].

As for line 3, recall that G is a subset of $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$. Furthermore, the sole purpose of G is to compute $R_B = R \setminus G$. We now observe that $\neg G$, the complement of G, is as good as G. In fact, $R_B = R \cap (\neg G)$. So by considering $\neg G$ instead, what we are looking for is an overapproximation of $\neg(gfp\ \lambda Y.\, W \cap \tilde{g}(Y))$. Next we recall Park's theorem replacing the above expression by a least fixpoint expression.

Theorem 3 (From [20]). *Let $\langle L, \sqsubseteq, \bigsqcap, \bigsqcup, \top, \bot, \neg \rangle$ be a complete Boolean algebra and let $f \in L \to L$ be an order-preserving function then $f' = \lambda X.\, \neg(f(\neg X))$ is an order-preserving function on L and $\neg(gfp\ f) = lfp\ f'$.*

Park's theorem applies in our setting because computations are carried over the Boolean algebra $\langle 2^{(Q \times Q)}, \subseteq, \cap, \cup, (Q \times Q), \emptyset, \neg \rangle$. Applying it to $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ where $\tilde{g}(Y) = \neg(\neg Y \circ R^{-1})$, we find that

$$\neg(gfp\ \lambda Y.\, W \cap \neg(\neg Y \circ R^{-1})) = lfp\ \lambda Y.\, (\neg W) \cup Y \circ R^{-1} \quad .$$

Therefore, to implement line 3, we rely on abstract interpretation to compute an over-approximation of $lfp\ \lambda Y.\, (\neg W) \cup Y \circ R^{-1}$, hence, by negation, an underapproximation of $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ therefore complying with the requirement on G.

As far as abstract interpretation is concerned, our implementation uses a combination of predicate abstraction [17] and case splitting.The set of predicates is given by a finite

set of atomic linear constraints and is also closed under negation, e.g., if $x + y \geq 0$ is a predicate then $x + y \leq -1$ is also a predicate. Abstract values are positive Boolean combination of atoms taken from the set of predicates. Observe that although negation is forbidden in the definition of abstract values, the abstract domain is closed under complement.

The set of predicates is chosen so as the following invariant to hold: each time the control hits line 3, the set contains enough predicates to represent precisely each well-founded relation in W. Our implementation provides enhanced precision by enforcing a stronger invariant: besides the above predicates for W, it includes all atomic linear constraints occurring in the formulas representing $X_1, \ldots, X_\ell$ where $\ell \geq 0$, $X_0 = (\neg W)$ and $X_{i+1} = (\neg W) \cup X_i \circ R^{-1}$. The value of ℓ is user-defined and, in our experiments, it did not exceed 1.

To further enhance precision at line 3, we apply case splitting. The set of R-traces is partitioned using the linear atomic constraints of the form $f(\bar{x}') < f(\bar{x})$ that appear in W. More precisely, partitioning R on $f(\bar{x}') < f(\bar{x})$ is done by replacing each ρ_i by $(\rho_i \wedge f(\bar{x}') < f(\bar{x})) \vee (\rho_i \wedge f(\bar{x}') \geq f(\bar{x}))$.

As for conditional termination, overapproximating $\mathcal{Z} = gfp\ \lambda\mathcal{X}.\, pre[R_B](\mathcal{X})$ is done by computing the last element $\mathcal{X}_\ell$ from the finite sequence $\mathcal{X}_0, \ldots, \mathcal{X}_\ell$ given by $\mathcal{X}_0 = Q$ and $\mathcal{X}_{i+1} = \mathcal{X}_i \wedge pre[R_B](\mathcal{X}_i)$ where ℓ is predefined. The result is always representable as DNF formula where the atoms can be any atomic linear constraints. As for $\mathcal{V} = lfp\ \lambda\mathcal{X}.\, \mathcal{X}_\ell \cup pre[R](\mathcal{X})$, an overapproximation is computed in a similar way to that of line 3, i.e., using a combination of predicate abstraction and case splitting.

7 Experiments

We have evaluated our prototype implementation against a set of benchmarks collected from publications in the area [9,8]. In what follows, we present the results of our implementation for those loops, and compare them to existing tools for proving termination [25,8,7] as well as tools for inferring preconditions for termination [9]. We compare the different techniques according to what the corresponding implementations report. We ignore performance because, for the selected benchmarks, little insight can be gained from performance measurements when an implementation was available (which was not always the case [26]).

The benchmarks accompanied with our results are depicted in Table 1. Translating each loop to a relation of the form $R = \rho_1 \vee \cdots \vee \rho_n$ is straightforward. Every line in the table includes a loop and its inferred termination precondition (*true* means it terminates for any input). In addition, preconditions (different from *true*) marked with • are optimal, i.e., the corresponding loop is non-terminating for any state in the complement.

We have divided the benchmarks into 3 groups: (1–5), (6–15) and (16–41). With the exception of loop 1, each loop in group (1–5) includes non-terminating executions and thus those loops are suitable for inferring preconditions. Our implementation reports the same preconditions as the tool of Cook et al. [9] save for loop 1 for which their tool is reported to infer the precondition $x > 5 \vee x < 0$, while we prove termination for all input. Note that every other tool used in the comparison [8,7,25] fail to prove termination of this loop. Further, the precondition we infer for loop 5 is optimal.

Table 1. Benchmarks used in experiments. Loops (1–5) are taken from [9] and (6–41) from [8]

♯	loop		termination precondition
1	`while (x≥0) x'=-2x+10;`		*true*
2	`while (x>0) x'=x+y; y'=y+z;`		$x{\leq}0 \vee z{<}0 \vee$ $(z{=}0 \wedge y{<}0) \vee$ $x{+}y{\leq}0 \vee x{+}2y{+}z{\leq}0 \vee$ $x{+}3y{+}3z{\leq}0$
3	`while (x≤N)` `  if (*) { x'=2*x+y; y'=y+1; } else x'=x+1;`		$x > n \vee x + y \geq 0$

4	`@requires n>200 and y<9`
	`while (1)`
	`  if (x<n) {`
	`  x'=x+y;`
	`  if (x'≥200) break;`
	`  }`
			$n \leq 200 \vee y \geq 9 \vee$ $(x < n \wedge y \geq 1) \vee$ $(x{<}n \wedge x{\geq}200 \wedge x{+}y{\geq}200)$

5	`while (x<>y) if (x>y) x'=x-y; else y'=y-x;`	•	$(x \geq 1 \wedge y \geq 1) \vee x = y$
6	`while (x<0) x'=x+y; y'=y-1;`		$x \geq 0 \vee x + y \geq 0 \vee$ $x + 2y \geq 1 \vee x + 3y \geq 3$
7	`while (x>0) x'=x+y; y'=-2y;`	•	$x \leq 0 \vee y \neq 0$
8	`while (x<y) x'=x+y; y'=-2y;`	•	$x \geq 0 \vee y \neq 0$
9	`while (x<y) x'=x+y; 2y'=y;`	•	$x \geq 0 \vee y \neq 0$
10	`while (4x-5y>0) x'=2x+4y; y'=4x;`	•	$5y - 4x \geq 0 \vee$ $(3x{-}4y \geq 0 \wedge 16x{-}21y \geq 1)$
11	`while (x<5) x'=x-y; y'=x+y;`	•	$x \neq 0 \vee y \neq 0$
12	`while (x>0 and y>0) x'=-2x+10y;`	•	$x \leq 3 \vee 10y - 3x \neq 0$
13	`while (x>0) x'=x+y;`		$x \leq 0 \vee y < 0 \vee x + y \leq 0$
14	`while (x<10) x'=-y; y'=y+1;`	•	$y \leq -10 \vee x \geq 10$
15	`while (x<0) x'=x+z; y'=y+1; z'=-2y`		$x \geq 0 \vee x + z \geq 0$
16	`while (x>0 and x<100) x'≥2x+10;`	★	*true*
17	`while (x>1) -2x'=x;`	★	*true*
18	`while (x>1) 2x'≤x;`	★	*true*
19	`while (x>0) 2x'≤x;`	★	*true*
20	`while (x>0) x'=x+y; y'=y-1;`		*true*
21	`while (4x+y>0) x'=-2x+4y; y'=4x;`		$4x + y \leq 0 \vee$ $(x - 4x \geq 0 \wedge 8x - 15y \geq 1)$
22	`while (x>0 and x<y) x'=2x; y'=y+1;`		*true*
23	`while (x>0) x'=x-2y; y'=y+1;`		*true*
24	`while (x>0 and x<n) x'=-x+y-5; y'=2y; n'=n;`		*true*
25	`while (x>0 and y<0) x'=x+y; y'=y-1;`	★	*true*
26	`while (x-y>0) x'=-x+y; y'=y+1;`		*true*
27	`while (x>0) x'=y; y'=y-1;`		*true*
28	`while (x>0) x'=x+y-5; y'=-2y;`		*true*
29	`while (x+y>0) x'=x-1; y'=-2y;`		*true*
30	`while (x>y) x'=x-y; 1≤y'≤2`	★	*true*
31	`while (x>0) x'=x+y; y'=-y-1;`		*true*
32	`while (x>0) x'=y; y'≤-y;`	★	*true*
33	`while (x<y) x'=x+1; y'=z; z'=z;`		*true*
34	`while (x>0) x'=x+y; y'=y+z; z'=z-1;`		*true*
35	`while (x+y≥0 and x≤z) x'=2x+y; y'=y+1; z'=z`		*true*
36	`while (x>0 and x≤z) x'=2x+y; y'=y+1; z'=z`		*true*
37	`while (x≥0) x'=x+y; y'=z; z'=-z-1;`		*true*
38	`while (x-y>0) x'=-x+y; y'=z; z'=z+1;`		*true*
39	`while (x>0 and x<y) x'>2x; y'=z; z'=z;`		*true*
40	`while (x≥0 and x+y≥0) x'=x+y+z; y'=-z-1; z'=z;`	★	*true*
41	`while (x+y≥0 and x≤n) x'=2x+y; y'=z; z'=z+1; n'=n;`		*true*

All the loops (6–15) are non-terminating. Chen et al. [8] report that their tool cannot handle them since it aims at proving termination and not inferring preconditions for termination. We infer preconditions for all of them, and in addition, most of them are optimal (those marked with •). Unfortunately for those loops we could not compare with the tool of Cook et al. [9], since there is no implementation available [26].

Loops in the group (16–41) are all terminating. Those marked with ⋆ actually have linear ranking functions, those unmarked require disjunctive well-founded transition invariants with more than one disjunct. We prove termination of all of them except loop 21. We point that the tool of Chen et al. [8] also fails to prove termination of loop 21, but also of loop 34. On the other benchmarks, they prove termination. They also report that PolyRank [7] failed to prove termination of any of the loops that do not have a linear ranking function. In addition, we applied ARMC [25] on the loops of the group (16–41). ARMC, a transition invariants based prover, succeeded to prove termination for all those loops with a linear ranking function (marked with ⋆) and also loop 39.

Next we discuss in details the analysis of two selected examples from Table 1.

Example 8. Let us explain the analysis of loop 1 in details starting with the root call $\texttt{Acabar}(R, \emptyset)$ where $R = \{x \geq 0, x' = -2x + 10\}$. At line 2, since R includes the bound $x \geq 0$, i.e., $f(x) = x$ is a potential linear ranking function, we add $\{x' < x, x \geq 0\}$ to W. Computing G at line 3, hence R_B at the following line, results in $R_B = \rho_1 \vee \rho_2$ where $\rho_1 = \{x' = -2x + 10, x \geq 0, x \leq 3\}$ and $\rho_2 = \{x' = -2x + 10, x \geq 4, x \leq 5\}$.

Note that ρ_1 is enabled for $0 \leq x \leq 3$ and in this case $x' > x$. Also ρ_2 is enabled for $x = 4$ or $x = 5$ for which $x' < x$ and thus $\rho_2 \subseteq W$, however, after one more iteration, the value of x increases (this is why ρ_2 is included in R_B). Transitions for which $x > 5$ are not included in R_B, hence they belong to R_G itself included in W (Lem. 3). Hence when $x > 5$ termination is guaranteed, this is also easily seen since those transitions terminate after one iteration.

Since R_B is neither empty nor equal to R, a recursive call to $\texttt{Acabar}(R_B, W)$ takes place. At line 2, we add $\{-x' < -x, 10 - x \geq 0\}$ to W since $f(x) = 10 - x$ is a linear ranking function for ρ_1. Note that ρ_2 has the linear ranking function $f(x) = x$ already included in W. Computing G at line 3, hence R_B, yields $R_B = \emptyset$ and therefore we conclude that the loop terminates for any input. ■

Example 9. Let us explain the analysis of loop 9 in details starting with the root call $\texttt{Acabar}(R, \emptyset)$ where $R = \{x < y, x' = x + y, 2y' = y\}$. At line 2, since R includes the bound $y - x > 0$, i.e., $f(x, y) = y - x - 1$ is a potential linear ranking function, we add $\{y' - x' < y - x, y - x - 1 \geq 0\}$ to W. Computing G at line 3, hence R_B yields $R_B = \{x < y, x' = x + y, 2y' = y, y \leq 0\}$. Note that R_B exclusively consists of transitions where y is not positive, in which case $x' - y' \geq x - y$ and thus not included in W. Transitions where y is positive are not included in R_B (hence they belong to R_G) since they always decrease $x - y$, and thus are transitively included in W (Lem. 3).

Since R_B is neither empty nor equal to R, we call recursively $\texttt{Acabar}(R_B, W)$. At line 2, since R includes the bound $y \leq 0$ (or equivalently $-y \geq 0$), i.e., $f(x, y) = -y$ is a potential linear ranking function, we add $\{-y' < -y, -y \geq 0\}$ to W. Computing G at line 3, hence R_B yields $R_B = \{x < y, x' = x + y, 2y' = y, y = 0\}$. Note that R_B exclusively consists of transitions where $y = 0$, which keeps both values of x and y unchanged.

Transitions in which y is negative belong to R_G, hence they are transitively covered by W (Lem. 3), in particular by the last update (viz. $\{-y' < -y, -y \geq 0\}$) to W.

Since R_B is neither empty nor equal to R, we call recursively `Acabar`(R_B, W). This time our implementation does not further enrich W with a well-founded relation, and as a consequence, after computing G at line 3, we get that $R_B = R$. Hence, `Acabar` returns with $R_B = \{x < y, x' = x + y, 2y' = y, y = 0\}$.

Now, given R_B, we infer a precondition for termination as described in Sec. 5. We first compute $gfp\ \lambda\mathcal{X}.\, pre[R_B](\mathcal{X})$, which in this case, converges in two steps with $\mathcal{Z} \equiv y = 0 \wedge x < 0$. Then we compute $lfp\ \lambda\mathcal{X}.\, \mathcal{Z} \cup pre[R](\mathcal{X})$, which results in $\mathcal{V} \equiv y = 0 \wedge x < 0$. The complement, $\mathcal{P} \equiv y < 0 \vee y > 0 \vee x < 0$, is a precondition for termination. Note that the result is optimal, i.e., $\mathcal{V}$ is a precondition for non-termination. Optimality is achieved because $\mathcal{Z}$ and $\mathcal{V}$ coincide with the *gfp* and the *lfp* of the corresponding operators, and are not overapproximations. ■

8 Conclusion

This work started with the invited talk of A. Podelski at ETAPS '11 who remarked that the inclusion check $R^+ \subseteq W$ is equivalently formulated as a safety verification problem where states are made of pairs. Back to late 2007, a PhD thesis [16] proposed a new approach to the safety verification problem in which the author shows how to leverage the equivalent backward and forward formulations of the inclusion check. Those two events planted the seeds for the backward inclusion check $R \subseteq W^-$, and later `Acabar`.

Initial States. For the sake of simplicity, we deliberately excluded the initial states $\mathcal{I}$ from the previous developments. Next, we introduce two possible options to incorporate knowledge about the initial states in our framework. The first option consists in replacing R by R' that is given by $R \cap (Acc \times Acc)$ where Acc denotes (an overapproximation of) the reachable states in the system. Formally, Acc is given by the least fixpoint $lfp\ \lambda\mathcal{X}.\, \mathcal{I} \cup post[R](\mathcal{X})$.

The second option is inspired by the work of Cousot [12] where he mixes backward and forward reasoning. We give here some intuitions and preliminary development. Recall that the greatest fixpoint $gfp\ \lambda Y.\, W \cap \tilde{g}(Y)$ of line 3 is best understood as the result of removing all those pairs $(s, s') \in W$ such that $(s, s') \circ R^+ \not\subseteq W$. We observe that the knowledge about initial states is not used in the greatest fixpoint. A way to incorporate that knowledge is to replace the greatest fixpoint expression by the following one $gfp\ \lambda Y.\, (B \cap W) \cap \tilde{g}(Y)$ where B takes the reachable states into account. In a future work, we will formally develop those two options and evaluate their benefit.

Related Works. As for termination, our work is mostly related to the work of Cook et al. [10,11] where the inclusion check $R^+ \subseteq W$ [22] is put to work by incrementally constructing W. Our approach, being based on the dual check $R \subseteq W^-$, adds a new dimension of modularity/incrementality in which R is also modified to safely exclude those transitions for which the current proof is sufficient. The advantage of the dual check was shown experimentally in Sec. 7. However, let us note that in our implementation we use potential ranking functions and case splitting, which are not used in ARMC [10]. Moreover, it smoothly applies to conditional termination.

Kroening et al. [18] introduced the notion of compositional transition invariants, and used it to develop techniques that avoid the performance bottleneck of previous approaches [11]. Recently, Chen et al. [8] proposed a technique for proving termination of single-path linear-constraint loops. Contrary to their techniques, we handle general transition relations and our approach applies also to conditional termination. Alias et al. [3] developed a termination analysis for flowcharts by incrementally synthesizing a lexicographical (linear) ranking function. As we do, they discard transitions covered by the current ranking function. They differ from us in the granularity by considering all the transitions corresponding an edge in the flowchart. On the contrary, our reasoning is independent from the system description. As for conditional termination, the work of Cook et al. [9] is the closest to ours. However, we differ in the following points: (*a*) we do not use universal quantifier elimination, whose complexity is usually very high, depending on the underlying theory used to specify R. Instead, we adapt a fixpoint centric view that allows using abstract interpretation, and thus to control precision and performance; (*b*) we do not need special treatment for loop with phase transitions (as the one of Sec. 2), they are handled transparently in our framework. Podelski et al. [23] studied the problem of conditional termination for heap manipulating programs. In this context, the inferred conditions are assumptions on the heap (reachability, aliasing, etc.). Bozga et al. [6] studied the problem of deciding conditional termination. Their main interest is to identify family of systems for which $gfp\ \lambda X.\, pre[R](X)$, the set of non-terminating states, is computable.

It is worth "terminating" by mentioning an alternate formulation of the termination check $R^+ \subseteq W$ [19]. Works based on this alternate formulation, in particular those that construct global ranking functions for R [4], might serve as a starting point to understand some (completeness) properties of our approach. This is left for future work.

References

1. Acabar, `http://loopkiller.com/acabar`
2. Albert, E., Arenas, P., Genaim, S., Puebla, G., Zanardini, D.: COSTA: Design and implementation of a cost and termination analyzer for java bytecode. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2007. LNCS, vol. 5382, pp. 113–132. Springer, Heidelberg (2008)
3. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
4. Ben-Amram, A.M.: Size-change termination, monotonicity constraints and ranking functions. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 109–123. Springer, Heidelberg (2009)
5. Ben-Amram, A.M., Genaim, S.: On the linear ranking problem for integer linear-constraint loops. In: POPL 2013: Proc. 40th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 51–62. ACM (2013)
6. Bozga, M., Iosif, R., Konečný, F.: Deciding conditional termination. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 252–266. Springer, Heidelberg (2012)
7. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)

8. Chen, H.Y., Flur, S., Mukhopadhyay, S.: Termination proofs for linear simple loops. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 422–438. Springer, Heidelberg (2012)
9. Cook, B., Gulwani, S., Lev-Ami, T., Rybalchenko, A., Sagiv, M.: Proving conditional termination. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 328–340. Springer, Heidelberg (2008)
10. Cook, B., Podelski, A., Rybalchenko, A.: Abstraction refinement for termination. In: Hankin, C., Siveroni, I. (eds.) SAS 2005. LNCS, vol. 3672, pp. 87–101. Springer, Heidelberg (2005)
11. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI 2006: Proc. 27th ACM-SIGPLAN Conf. on Programming Language Design and Implementation, pp. 415–426. ACM (2006)
12. Cousot, P.: Méthodes Itératives de construction et d'approximation de points fixes d'opérateurs monotones sur un treillis, analyse sémantique de programmes. Thèse d'état ès sciences mathématiques, Université scientifique et médicale de Grenoble (March 1978) (in French)
13. Cousot, P.: Partial completeness of abstract fixpoint checking, invited paper. In: Choueiry, B.Y., Walsh, T. (eds.) SARA 2000. LNCS (LNAI), vol. 1864, pp. 1–25. Springer, Heidelberg (2000)
14. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL 1977: Proc. 4th ACM SIGACT-SIGPLAN Symp. on Principles of Programming Languages, pp. 238–252. ACM (1977)
15. Davey, B.A., Priestley, H.A.: Introduction to Lattices and Order. CUP, Cambridge (1989)
16. Ganty, P.: The Fixpoint Checking Problem: An Abstraction Refinement Perspective. Ph.D. thesis, Université Libre de Bruxelles (2007)
17. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
18. Kroening, D., Sharygina, N., Tsitovich, A., Wintersteiger, C.M.: Termination analysis with compositional transition invariants. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 89–103. Springer, Heidelberg (2010)
19. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL 2001: Proc. 28th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages, pp. 81–92. ACM (2001)
20. Park, D.: Fixpoint induction and proofs of program properties. In: Machine Intelligence, vol. 5, pp. 59–78. American Elsevier (1969)
21. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
22. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS 2004: Proc. 19th Annual IEEE Symp. on Logic in Computer Science, pp. 32–41. IEEE (2004)
23. Podelski, A., Rybalchenko, A., Wies, T.: Heap assumptions on demand. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 314–327. Springer, Heidelberg (2008)
24. Ramsey, F.P.: On a problem of formal logic. London Math. Society 30, 264–286 (1929)
25. Rybalchenko, A.: Armc (2008), `http://www7.in.tum.de/~rybal/armc/`
26. Rybalchenko, A.: Personal communication (2012)
27. Spoto, F., Mesnard, F., Payet, É.: A termination analyzer for java bytecode based on path-length. ACM Trans. Program. Lang. Syst. 32(3) (2010)

Better Termination Proving through Cooperation

Marc Brockschmidt[1], Byron Cook[2,3], and Carsten Fuhs[3]

[1] RWTH Aachen University
[2] Microsoft Research Cambridge
[3] University College London

Abstract. One of the difficulties of proving program termination is managing the subtle interplay between the finding of a termination argument and the finding of the argument's supporting invariant. In this paper we propose a new mechanism that facilitates better cooperation between these two types of reasoning. In an experimental evaluation we find that our new method leads to dramatic performance improvements.

1 Introduction

When proving program termination we are simultaneously solving two problems: the search for a termination argument, and the search for a supporting invariant. Consider the following example:

```
y := 1;
while x > 0 do
    x := x − y;
    y := y + 1;
done
```

To prove termination of this program we are looking to find both a termination argument (*i.e.*, "x decreases until 0") and a supporting invariant (*i.e.*, $y > 0$). The two are interrelated: Without $y > 0$, we cannot prove the validity of the (safety) property "x decreases until 0"; and without "x decreases towards 0", how would we know that we need to prove $y > 0$?

Several program termination proving tools (*e.g.* [15], [16], [23], [34], [39]) address this problem using a strategy that oscillates between calls to an off-the-shelf safety prover (*e.g.* [1], [4], [11], [26], [31], *etc.*) and calls to a rank function synthesis tool (*e.g.* [2], [7], [8], [35], *etc.*). In this setting a candidate termination argument is iteratively constructed. The safety prover proves or disproves the validity of the current argument via the search for invariants. Refinement of the current termination argument is performed using the output of a rank function synthesis tool when applied to counterexamples found by the safety prover.

A difficulty with this approach is that currently, the underlying tools do not share enough information about the overall state of the termination proof. For example, the rank function synthesis tool is only applied to the single path through

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 413–429, 2013.

the program described by the counterexample found by the safety prover, while the context of this single path is not considered at all. Meanwhile, the safety prover is unaware of things such as which paths in the program have already been deemed terminating and how those paths might contribute to other potentially infinite executions. The result is lost performance, as the underlying tools often make choices inappropriate to the common goal of fast termination proving.

In this paper we introduce a technique that facilitates cooperation between the underlying tools in a termination prover, thus allowing for decisions more appropriate to the common good of proving program termination. The idea is to use a single representation of the state of the termination proof search—called a *cooperation graph*—that both tools operate over. Nodes in the graph are marked as either termination-nodes or safety-nodes, thus indicating the role they play in the state of the proof. With this additional information exposed, we can now represent the progress of the termination proof search by modifying the termination subgraph. This has practical advantages. For example, the safety prover can be encouraged not to explore parts of the program that have already been proven terminating. On the rank function synthesis side, we can make use of the full program structure in order to find better termination arguments.

Our approach results in dramatic performance improvements compared to earlier methods and our implementation succeeds on numerous programs on which previous tools fail. In cases where previous tools do succeed, our implementation increases performance by orders of magnitude.

Related Work. Numerous tools and techniques exist for termination proving (*e.g.* [5], [7], [8], [10], [15], [17], [20], [21], [29], [34], [39], *etc.*). In many instances our approach is related but essentially incomparable with these previous tools. For example, size-change termination proving [29] sacrifices precision for consistency with a fixed *a priori* finite abstraction and an essentially fixed termination argument. The result is an analysis that will fail to prove termination in more complex cases, but that itself always terminates. This is in contrast to our technique which privileges precision over predictability (*e.g.* we use possibly non-terminating techniques during the search for supporting invariants).

The tools most similar to our own are ARMC [34], TRex [25], CProver [39], HSF [23], Terminator [15], and T2 [16]. As discussed above, the key difference here is in our treatment of shared information. These previous tools share only simple paths with the rank function synthesis procedure, and only the termination argument with the safety-based validity proving procedure. Our cooperation graph, while similar in principle to previous representations (*e.g.* [15]), exposes information in a way that facilitates operations on the graph that would have been difficult or unsound in previous approaches. To see the difference, we look to the experimental results which show a dramatic improvement over previous approaches when our technique is applied.

In order to make use of the information that we have exposed we borrow several existing techniques. For example, we adapt a program simplification strategy from the dependency pair framework [3,22,27] to our shared graph as a way

of recording lemmas during the proof search. We use a recently developed technique for efficient rank function synthesis with multiple control-flow locations [2]. Finally, we build upon a recently developed iterative method for finding lexicographic rank functions [16]. Our cooperation graphs facilitate the combination of these complementary techniques, leading to a new tool that outperforms all of the previous approaches.

Limitations. While in theory our approach works in a general setting, in our implementation we are focusing on sequential arithmetic programs (*e.g.* these programs do not use the heap or bitvectors). In some cases we have soundly abstracted C programs with heap to arithmetic programs (*e.g.* using a technique due to Magill *et al.* [30]); in other cases, as is standard in many tools (*e.g.* SLAM [4]), we essentially ignored bitvectors and the heap. Techniques that more accurately and efficiently reason about mixtures of heap and arithmetic are an area of open interest.

2 Example

We illustrate our approach using the example in Fig. 1, which displays a bubble-sort like program (the manipulation of the data has been abstracted away). In our setting we use a graph—called a *cooperation graph*—to facilitate sharing of information between a safety prover and a rank function synthesis procedure. See Fig. 2 for the cooperation graph at the start of the proof search. Here we have essentially duplicated the loops in the original program, with non-deterministic transitions from one copy of the program to the other (*i.e.*, τ_4 and τ_5). After duplication, we apply a few known tricks: In the new copy of the program, we follow the approach of Biere *et al.* [6] by adding nodes (*i.e.*, ℓ_1^d and ℓ_2^d) and transitions to take a snapshot of variable values (*i.e.*, γ_1 and γ_2). The current values of variables $\mathsf{i}, \mathsf{j}, \mathsf{n}$ are stored in copies $\mathsf{i}^c, \mathsf{j}^c, \mathsf{n}^c$ and the flag cp_k is set to indicate that a snapshot was taken at location ℓ_k. Furthermore, new transitions to an error

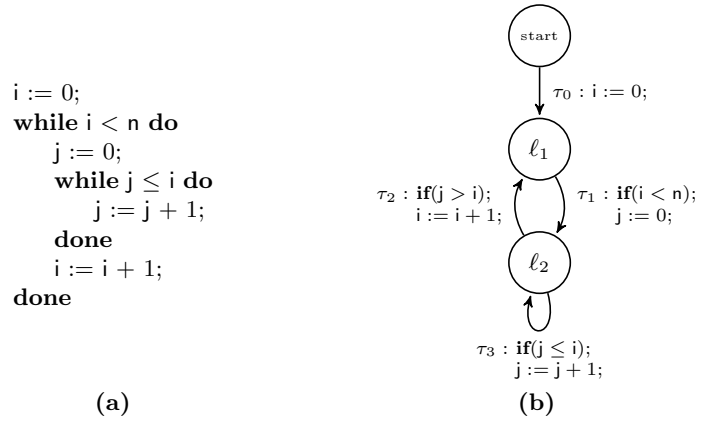


Fig. 1. Textual and control-flow graph representation of skeleton bubble sort routine

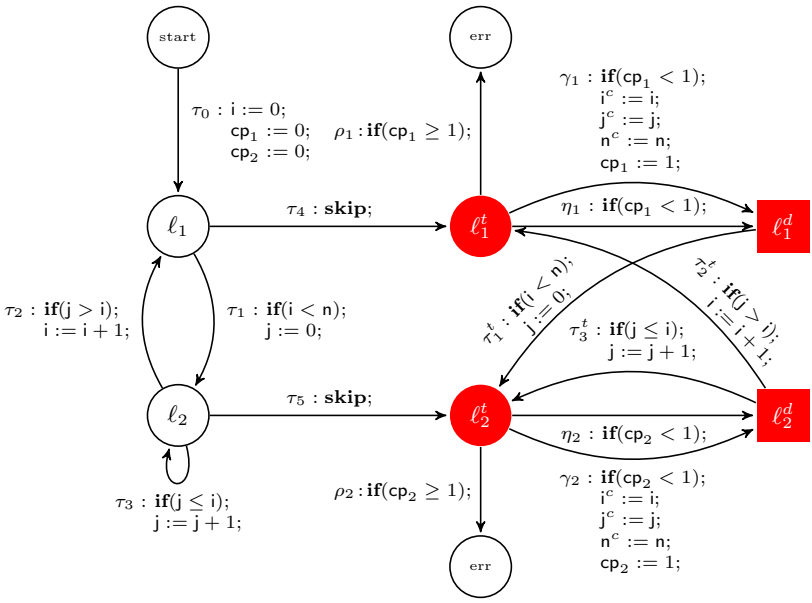


Fig. 2. Cooperation graph derived from Fig. 1

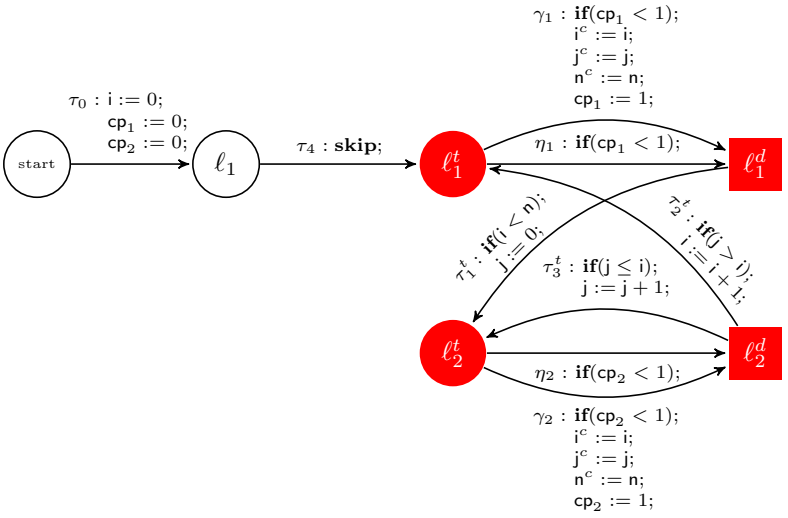
location "err" have been added that—using the approach of Cook *et al.* [15]—can be strengthened later by partial termination arguments. Proving this error location unreachable then implies a termination proof for the input program. In the resulting graph, reasoning about termination is performed on the right-hand side—called the termination subgraph—by a procedure built around an efficient rank function synthesis. The search for supporting invariants is performed on the left-hand side—called the safety subgraph—by a safety prover.

The advantage of the duplication (*i.e.* the termination and safety subgraphs) is that we can easily restrict certain operations to either subgraph, but we maintain a connection between them. We use the safety subgraph to describe an over-approximation of all reachable states, while the termination subgraph is an over-approximation of those states for which termination has not been proven yet. This allows us to perform operations in the one half that may not make sense (or may be unsound) in the other. For example, when we prove that transitions in the termination subgraph can only be used finitely often, we can simply remove them, as they cannot contribute to infinite executions. This is only sound because the safety subgraph remains unchanged in this simplification, which keeps the set of reachable states unchanged and hence allows reasoning about safety/invariants. In our setting, these iterative program simplifications encode the progress of the termination proof search and are directly available to the safety prover when searching for more counterexamples.

The structure of the graph guides the safety prover to unproven parts of the program, directly yielding relevant counterexamples that can be used by the rank function synthesis to produce better termination arguments. If these do

not allow a program simplification, they still guide the generation of invariants by the safety prover for nodes in the safety subgraph. These in turn then support reasoning about the validity of termination arguments in the termination subgraph.

Termination Proof Sketch. We now illustrate how termination is proved in our setting. We begin searching for a path from the "start" location to the error location "err". We might, for example, choose the path $\langle \tau_0, \tau_4, \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t, \rho_1 \rangle$ where τ_0 is drawn from the safety subgraph and the other transitions come from the termination subgraph. Here, $\langle \gamma_1, \tau_1^t, \eta_2, \tau_3^t, \eta_2, \tau_2^t \rangle$ form a cycle in the execution, returning back to location ℓ_1^t. In our approach we do not simply use this command sequence directly to search for a new termination argument (as is done in previous tools). Instead, we additionally consider all transitions from the termination subgraph that enter and exit nodes in the strongly connected component (SCC) containing the found cycle of termination-transitions in the counterexample. We call this enclosing SCC the *SCC context* of a certain cycle. In this case, because the graph is so small, this includes the entire termination subgraph:



By examining a graph that includes extra termination-edges (*e.g.* τ_3^t) we can see that the rank function $\mathsf{n} - \mathsf{i}$ is a better rank function than $\mathsf{j} - \mathsf{i}$ because τ_3^t modifies j. Without τ_3^t, j appears as a constant and hence, $\mathsf{j} > \mathsf{i}$ looks like a suitable candidate invariant supporting the termination argument $\mathsf{j} - \mathsf{i}$.

Fig. 3 is the state of the cooperation graph after considering one counterexample. We use the rank function with $\mathsf{n} - \mathsf{i} + 1$ for both ℓ_1^t and ℓ_1^d, and $\mathsf{n} - \mathsf{i}$ for both ℓ_2^t and ℓ_2^d. The value of this rank function is decreasing each time we use the transition τ_1^t, and the condition $\mathsf{i} < \mathsf{n}$ implies that the rank function is bounded from below. Hence, τ_1^t can only be used finitely often and we can remove it from the termination subgraph. Removing this transition is helpful

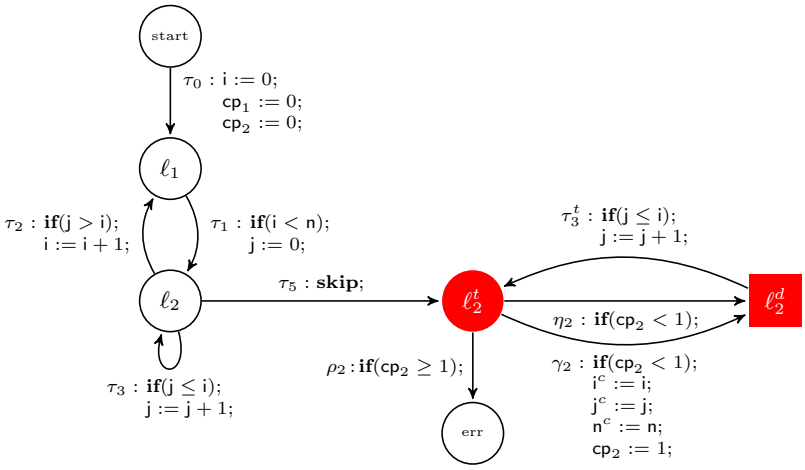


Fig. 3. Cooperation graph after safety and termination analysis on the graph from Fig. 2. Due to termination analysis, the transition τ_2^t has been removed. Afterwards, ℓ_1^t was not part of a non-trivial SCC anymore, so it, its duplicate ℓ_1^d, and the connecting transitions were removed.

for future iterations of the proof search, as it no longer needs to be considered when searching for further counterexamples. This also allows to remove ℓ_1^t, ℓ_1^d and all transitions connected to the two, as they are not part of a non-trivial strongly connected component anymore and hence cannot occur infinitely often in an execution. Because these nodes and transitions are only used to reason about termination, and not safety, we can soundly remove them. Removing the corresponding node ℓ_1 from the safety subgraph is unsound, as this would make the inner loop unreachable, without requiring a termination proof for it. In our setting we use an incremental implementation of lazy abstraction with interpolation (*à la* Impact [31]) to represent the inductive invariants for safety proving. We are not displaying the additional information inferred by this safety proving method in our cooperation graph here.

In the next iteration, starting on Fig. 3, all possible cycles allowed in the termination subgraph use the transition τ_3^t. This transition can easily be proved well-founded with the rank function $\mathsf{i} - \mathsf{j}$ for the locations ℓ_2^t and ℓ_2^d, allowing us to remove τ_3^t and then, ℓ_2^t, ℓ_2^d and all connected transitions, leaving us with a cooperation graph with an empty termination subgraph (*i.e.*, with an empty termination subgraph we are left with what is essentially the original graph from Fig. 1). Thus we have proved termination. (In practice, our algorithm in Fig. 5 handles simple examples such as this one already in a preprocessing step. Here we have given a counterexample-based termination proof for illustration purposes.)

3 Algorithm

In this section, we describe our new termination proving method more formally.

Preliminaries. We represent programs as graphs of program locations connected by transition rules with conditions and assignments to a set of integer variables $\mathcal{V}$. The canonical initial location is called *start*. Program states are tuples $(k, \boldsymbol{x})$, with k the current program location and $\boldsymbol{x}$ a column vector of the values of $\mathcal{V}$ in some fixed order. Transitions are labeled by formulas relating pre- and post-variables, where x' is the post-variable corresponding to the pre-variable x (resp. $\boldsymbol{x}$ are all pre-variables in fixed order, and $\boldsymbol{x'}$ the post-variables). The statement $\mathsf{i} := \mathsf{i} + 1$ is represented as $i' = i + 1$ and $\mathbf{if}(\mathsf{i} < \mathsf{n})$ as $i < n$. For example, the commands on transition τ_1 are equivalent to the formula $i < n \wedge i' = i \wedge j' = 0 \wedge n' = n$. In this paper, we only consider linear program transitions and hence use the constraint system $A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}$ instead of the corresponding formula.

A program execution is a possibly infinite sequence of program states $(k_1, \boldsymbol{x}_1)$, $(k_2, \boldsymbol{x}_2), \ldots$ with $k_1 = \mathit{start}$, $\boldsymbol{x}_1$ freely chosen and for each pair $(k_i, \boldsymbol{x}_i), (k_{i+1}, \boldsymbol{x}_{i+1})$, there is a program transition $(k_i, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k_{i+1})$ such that $A\left(\begin{smallmatrix}\boldsymbol{x}_i\\ \boldsymbol{x}_{i+1}\end{smallmatrix}\right) \geq \boldsymbol{a}$ holds. We call a program terminating if and only if it has no infinite execution.

Finding Termination Arguments. Past tools used constraint-based approaches for finding rank functions for (sub)programs involving only one program location (*e.g.* ARMC [34] and Terminator [15] use Podelski & Rybalchenko's rank function synthesis method [35], T2 [16] uses the approach due to Bradley *et al.* [7] for lexicographic rank functions). In our setting, we need to find rank functions for the SCC contexts of counterexamples in the termination subgraph, which might involve transitions over several program points. For this purpose we use the lexicographic rank function synthesis due to Alias *et al.* [2] to find linear rank functions for a set of transitions using possibly several program locations.

Given a (finite) set of program transitions $\mathcal{T}$, we prove termination iteratively. When proving that transitions cannot be used infinitely often in an infinite execution, we use an approach from the dependency pair framework [3,22,27] to remove them. For this, we choose a sequence of rank functions $f^1, \ldots, f^m$ that measure program states. A *$\mathcal{T}$-orienting rank function* f is a measure of program states in some well-founded ordered domain such that no transition $t \in \mathcal{T}$ allows an increase of this measure, *i.e.*, we require:

$$\bigwedge_{(k, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k') \in \mathcal{T}} \forall \boldsymbol{x}, \boldsymbol{x'}. A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a} \rightarrow f((k, \boldsymbol{x})) \geq f((k', \boldsymbol{x'})) \tag{1}$$

Furthermore, we want that for at least one of the transitions $t = (k, A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a}, k')$ in $\mathcal{T}$ the measure is actually decreasing and is bounded from below (0 is a minimal element in our domain):

$$\forall \boldsymbol{x}, \boldsymbol{x'}. A\left(\begin{smallmatrix}\boldsymbol{x}\\ \boldsymbol{x'}\end{smallmatrix}\right) \geq \boldsymbol{a} \rightarrow (f((k, \boldsymbol{x})) > f((k', \boldsymbol{x'})) \wedge f((k, \boldsymbol{x})) \geq 0) \tag{2}$$

Similar to the dependency pair framework [3,22,27] and to monotonicity constraints [12], we compose lexicographic termination arguments from such $\mathcal{T}$-orienting rank functions. If $\textsc{Decreasing}(\mathcal{T}, f) \subseteq \mathcal{T}$ is the set of transitions for which (2) holds for some f with (1), then for proving termination it suffices to consider executions that use only transitions from $\mathcal{T} \setminus \textsc{Decreasing}(\mathcal{T}, f)$ infinitely often (see also our technical report [9]).

Consequently, we construct lexicographic termination arguments for a set of transitions $\mathcal{T}$ by iteratively synthesizing such rank functions f. A transition $\delta \in \textsc{Decreasing}(\mathcal{T}, f)$ can only occur a finite number of times, so we ignore it for the rest of our termination proof and only consider suffixes of infinite executions that do not use δ anymore. In our cooperation graphs, we build upon this observation by removing transitions from the termination subgraph. There, any finite prefix of a computation can be represented using the (unchanged) safety subgraph, while the infinite suffix of a possibly non-terminating computation is represented by the simplified termination subgraph. By repeatedly removing transitions using different rank functions $f^1, \ldots, f^m$, we mirror the progress of building a lexicographic termination argument in the termination subgraph.

Cooperation Graphs. The procedure Instrument, from Fig. 4, is used to construct an initial cooperation graph with transitions $\mathcal{C}$ from a program $\mathcal{P}$ with locations $\mathcal{L}$ and transitions $\mathcal{T}$. We use two mappings SafetyLoc and TerminationLoc from $\mathcal{L}$ to fresh location names. We first create the safety subgraph of the cooperation graph as a copy of $\mathcal{P}$. For the termination subgraph, we first use SCC_Transitions to identify all transitions on components that may influence termination, *i.e.*, all non-trivial strongly connected components in the control-flow graph of $\mathcal{P}$, and copy these to the termination subgraph. We then connect the safety and termination subgraphs at cutpoints [19] of the original program, allowing a non-deterministic jump from the safety to the termination location.

We then apply the safety-reduction from Cook *et al.* [16] on cutpoints in the termination subgraph. The point of this reduction is to add an error location that is reachable *iff* the lexicographic termination argument is invalid. For this, we use a mapping CutpointDuplicate from cutpoints in the original program to fresh location names. We first "move" all transitions originally starting in the termination copy of the cutpoint p^t to its new duplicate. We then connect p^t to its duplicate by two transitions, one taking a snapshot of the current variable state, one doing nothing. In our example in Fig. 1, ℓ_1 is a cutpoint and we choose $\textsc{CutpointDuplicate}(\ell_1) = \ell_1^d$. The function snapshot produces the assignments needed to take a snapshot of the variables, *i.e.*, storing copies of variables v in an extra variable v^c and setting an integer flag cp_k that indicates that a snapshot at location k was taken. An example of the result is γ_1 from Fig. 2. Its twin nosnapshot does not do anything. Note that both resulting transitions can only be used if no snapshot of the program variables was taken at this program point before. Finally, we connect p^t to the error location by a transition that assumes that no decrease was found using the current set of rank functions. This set of rank functions is initially empty, and will be strengthened in the termination proof. Hence, the function nodecrease only returns the condition stating that a snapshot has been taken (*e.g.*, for ℓ_2^t we have $\mathsf{cp}_2 \geq 1$).

We define projections Safety and Termination on the cooperation graph. $\textsc{Safety}(\mathcal{C})$ are the transitions in $\mathcal{C}$ between locations in $\mathit{range}(\textsc{SafetyLoc})$, while the projection $\textsc{Termination}(\mathcal{C})$ are the transitions between locations in $\mathit{range}(\textsc{TerminationLoc}) \cup \mathit{range}(\textsc{CutpointDuplicate}) \cup \{\mathit{err}\}$. The safety projection $\textsc{Safety}(\mathcal{C})$ is isomorphic to the original program, and the termination

Input: Program with transitions $\mathcal{T}$, start location *start*
Output: Cooperation graph $\mathcal{C}$ with start location SafetyLoc(*start*)

1: $\mathcal{C} := \emptyset$
2: **for all** (ℓ, τ, ℓ') in $\mathcal{T}$ **do**
3: $\quad \mathcal{C} := \mathcal{C} \cup \{(\text{SafetyLoc}(\ell), \tau, \text{SafetyLoc}(\ell'))\}$
4: **end for**
5: **for all** (ℓ, τ, ℓ') in SCC_Transitions($\mathcal{T}$) **do**
6: $\quad \mathcal{C} := \mathcal{C} \cup \{(\text{TerminationLoc}(\ell), \tau, \text{TerminationLoc}(\ell'))\}$
7: **end for**
8: **for all** p in Cutpoints($\mathcal{T}$) **do**
9: $\quad p^t := \text{TerminationLoc}(p)$
10: $\quad p^d := \text{CutpointDuplicate}(p)$
11: $\quad \mathcal{C} := \mathcal{C} \cup \{(\text{SafetyLoc}(p), \textbf{skip}, p^t)\}$
12: $\quad$ **for all** (p^t, τ, ℓ') in $\mathcal{C}$ **do**
13: $\quad\quad \mathcal{C} := (\mathcal{C} \setminus \{(p^t, \tau, \ell')\}) \cup \{(p^d, \tau, \ell')\}$
14: $\quad$ **end for**
15: $\quad \mathcal{C} := \mathcal{C} \cup \{(p^t, \text{snapshot}(p), p^d), (p^t, \text{nosnapshot}(p), p^d)\}$
16: $\quad \mathcal{C} := \mathcal{C} \cup \{(p^t, \text{nodecrease}(p), err)\}$
17: **end for**
18: **return** $\mathcal{C}$

Fig. 4. Procedure Instrument, which initializes a new cooperation graph

projection corresponds to a termination problem without explicit start state.

In our termination proofs, Safety($\mathcal{C}$) represents the set of reachable states, and thus remains unchanged. In practice we use an incremental safety prover on this graph to find the necessary inductive invariants on demand. Meanwhile, Termination($\mathcal{C}$) represents the set of states for which we have not proven termination yet. We change Termination($\mathcal{C}$) in each iteration of the algorithm by possibly removing transitions and strengthening the conditions of the transitions from cutpoints to the error location. Consequently, questions of reachability and validity of invariants are based on the safety projection.

Refinement Algorithm. Our cooperation-based termination procedure is found in Fig. 5. We first use Instrument to create a cooperation graph from our input program. Then, we try to find (partial) lexicographic rank functions to simplify SCCs in the termination part of the graph, where Decreasing($\mathcal{S}, f$) identifies the transition rules satisfying (2) from above.

In simple examples such as that of Sect. 2, this preprocessing step can actually already prove termination, by removing all possible paths to the error location before the main loop begins. In more complex cases, we enter the main loop, in which we search for counterexamples to the decrease of the rank functions found so far. Our counterexamples are lassos, with the cycle part in the termination subgraph, starting in some cutpoint p, while the stem can always be represented using only the safety subgraph. In the cycle, we first take a snapshot of the current variable state and then return back to the termination copy of the cutpoint p, finding that the current set of rank functions do not show a decrease.

Input: Program with start state *start*, transitions $\mathcal{T}$
Output: "**Terminating**" or "**Unknown**"
1: $\mathcal{C} :=$ Instrument($\mathcal{T}$)
2: **for all** $\mathcal{S}$ in SCCs(Termination($\mathcal{C}$)) **do**
3: **while** $\exists$ $\mathcal{S}$-orienting rank function f **do**
4: $\mathcal{C} := \mathcal{C} \setminus$ Decreasing($\mathcal{S}, f$)
5: $\mathcal{S} := \mathcal{S} \setminus$ Decreasing($\mathcal{S}, f$)
6: **end while**
7: **end for**
8: **while** $\exists$ counterexample ($stem, cycle$) from SafetyLoc(*start*) to *err* in $\mathcal{C}$ **do**
9: $\mathcal{S} :=$ SCC_Context(Termination($\mathcal{C}$), *cycle*)
10: **if** $\exists$ $\mathcal{S}$-orienting rank function f **then**
11: $\mathcal{C} := \mathcal{C} \setminus$ Decreasing($\mathcal{S}, f$)
12: $\mathcal{C} :=$ Strengthen($\mathcal{C}, cycle, f$)
13: **else if** $\exists$ any rank function f for *cycle* **then**
14: $\mathcal{C} :=$ Strengthen($\mathcal{C}, cycle, f$)
15: **else**
16: **return** "Unknown"
17: **end if**
18: **end while**
19: **return** "Terminating"

Fig. 5. Procedure Refinement, which oscillates between a safety prover and a rank function synthesis tool using a cooperation graph

The counterexample is then used to synthesize a new rank function f. We first determine the SCC context of the cycle in the termination subgraph of the cooperation graph. We then try to find a rank function that is non-increasing for the SCC context of the cycle and decreases for our counterexample. If we find such a rank function, we remove any transitions that we have proven decreasing in all cases from the termination subgraph. We additionally use Strengthen to restrict the transition from the cutpoint p in the counterexample to the error location further, *i.e.*, we only allow going to the error location if the newly found rank function does not decrease.

The procedure Strengthen refines the partial termination argument in the safety-representation, as is done in previous tools (*e.g.* [15], [16], [23], [34], [25], [39]). We use lexicographic termination arguments as in Cook *et al.* [16]. Such an argument has the form $\langle f_1, \ldots, f_n \rangle$, where the f_i are the rank functions at some cutpoint p. If we can find a $\mathcal{S}$-orienting rank function, we can always prepend it to an existing lexicographic termination argument (computed "bottom-up"). If no such rank function could be found, we fall back to the method from Cook *et al.* [16], synthesizing a rank function such that a lexicographic termination argument for the counterexamples found so far can be constructed.

We then construct constraints that only allow a transition if an iteration did not make the variable state decrease w.r.t. this argument. Formally, we use the snapshots of old variable values to construct the pre-state $s = (p, \boldsymbol{x})$ at the beginning of the loop iteration and create the post-state $s' = (p, \boldsymbol{x'})$ of the loop

iteration from the current variables. Then, STRENGTHEN encodes the following condition:[1]

$$\neg\Big(\bigvee_{1\leq i\leq n} f_i(s) > f_i(s') \wedge f_i(s) \geq 0 \wedge \big(\bigwedge_{1\leq j<i} f_j(s) \geq f_j(s')\big)\Big)$$

STRENGTHEN uses the cycle passed as an argument to determine at which cut-point to strengthen the termination argument. If we could find no rank function for the cycle, we give up.[2] Finally, if no counterexamples exist anymore, we report termination. For a correctness proof of our termination proving procedure in Fig. 5, please see the technical report [9].

As in earlier work, this use of STRENGTHEN allows the termination prover to speculate termination arguments based on single counterexamples. The safety prover then has to find invariants proving the speculated argument to be correct, or provide more counterexamples. However, in our cooperation graph setting, the safety prover is helped in this by the removal of transitions proven to be terminating. This both speeds up the state space exploration and avoids to refute many spurious counterexamples. Moreover, by splitting executions into finite prefixes (in the safety subgraph) and possibly infinite suffixes (in the termination subgraph), the safety prover can infer "eventual invariants", *i.e.*, formulas that always hold after a finite time.

4 Evaluation

To evaluate the usefulness of our idea we have compared our implementation against the following tools/configurations:

- TERMINATOR [15], which implements an oscillation between rank function synthesis and safety using termination arguments expressed as transition invariants [36].
- T2 [16], which implements a TERMINATOR-like oscillation between rank function synthesis and safety using termination arguments expressed as lexicographic rank functions.
- COOPERATING-T2: Our implementation of the procedure from Fig. 5, which is based on T2.[3]
- ARMC [34], which also implements a TERMINATOR-like procedure. Note that, as of the writing of this paper, Rybalchenko's C-to-clauses converter was not complete, and thus we could not compare against HSF. Based on experience with the two tools we expect that ARMC and HSF will have comparable results when proving termination [38].
- APROVE [21], a termination prover based on the dependency pair framework [3,22,27] and including an implementation of the rank function synthesis *à la* Alias *et al.* [2]. APROVE does not generate invariants on demand and hence always uses the supporting invariant **true**.

[1] Disjunctions in transition conditions can be expressed using several transitions.

[2] Actually, we then attempt to prove non-termination, but the details of that procedure are orthogonal to the point of this paper.

[3] For details on accessing a source-based release of this tool, please see [9].

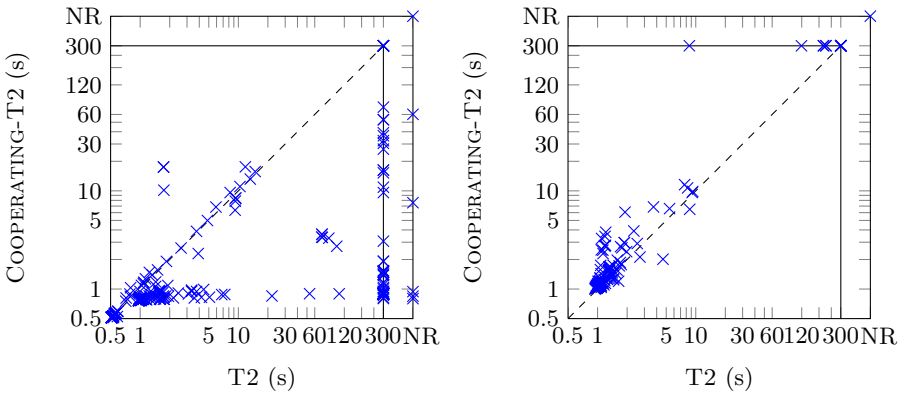


Fig. 6. Evaluation results of Cooperating-T2 vs. Standard T2, in log scale. Plot **(a)** represents the results of the two tools on terminating benchmarks, **(b)** represents non-terminating benchmarks. Timeout=300s. NR="No Result". The NR cases are due to failure of the underlying safety prover to find an inductive invariant.

- AProVE+Interproc, which uses the abstract interpretation tool Interproc [28] to generate as many invariants as possible using the Octagon abstract domain [33] before running AProVE.
- Size-change/MCNP, an implementation of termination proofs via monotonicity constraints [12], an efficient generalization of the size-change principle [29]. The abstraction from integer programs to monotonicity constraints is implemented in AProVE.
- KITTeL, another termination prover based on termination proving techniques from rewriting systems [18].

During our evaluation we ran tools on a set of 449 termination proving benchmarks drawn from a variety of applications that were also used in prior tool evaluations (*e.g.* Windows device drivers, the Apache web server, the PostgreSQL server, integer approximations of numerical programs from a book on numerical recipes [37], integer approximations of benchmarks from LLBMC [32] and other tool evaluations). Of these, 260 are known to be terminating and 181 are known to be non-terminating. For a handful examples, no result is known. These include the Collatz conjecture, and the remaining are very large and hence have not been analyzed manually. Our benchmarks and results can be found at

`http://verify.rwth-aachen.de/brockschmidt/Cooperating-T2/`

Experiments for Terminator, T2, and Cooperating-T2 were performed on a quadcore 2.26GHz E5520 system with 4GB of RAM and running Windows 7. All other experiments were performed on a quadcore 3.07GHz Core i7 system with 4GB of RAM and running Debian Linux 6. We ran all tools with a timeout of 300 seconds. When a tool returned early without a definite result or crashed, we display this in the plots using the special "NR" (no result) value.

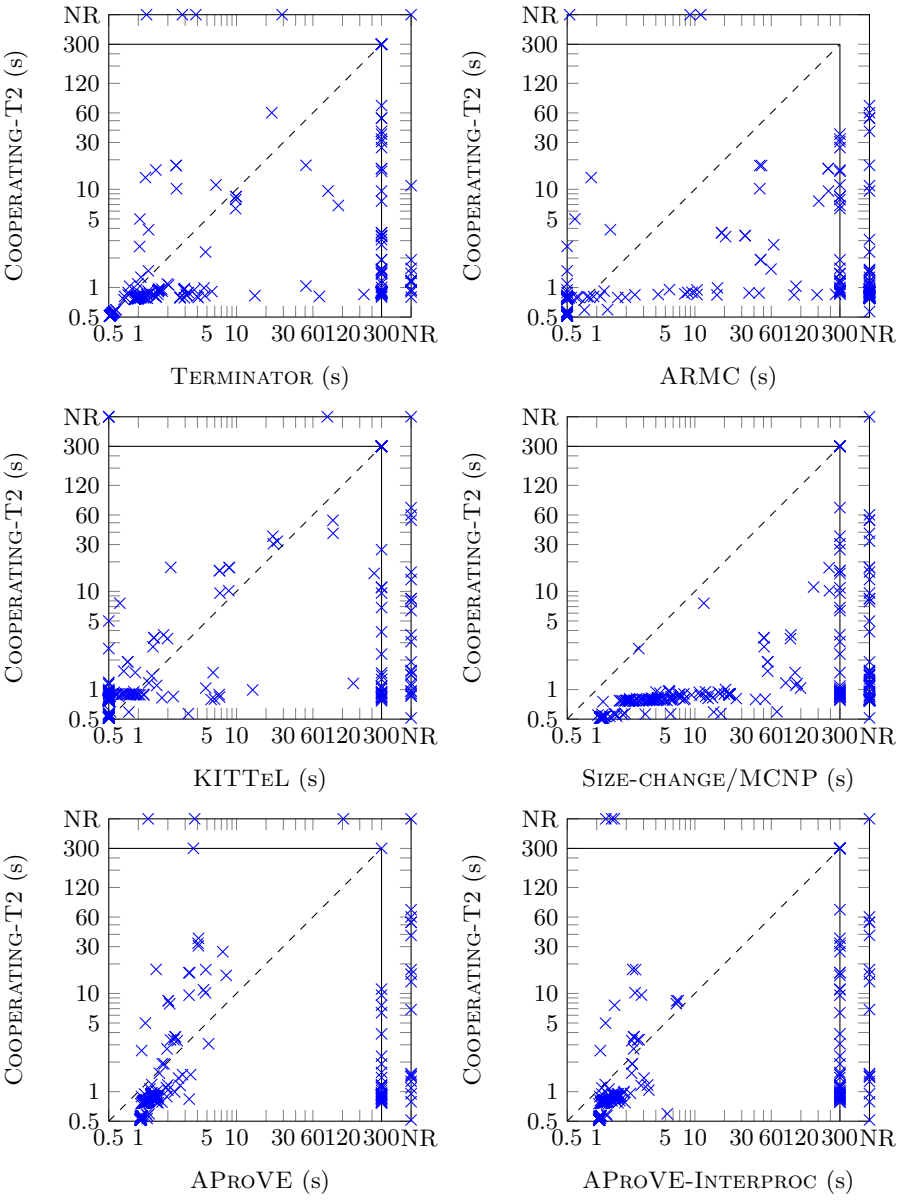


Fig. 7. Evaluation results of Cooperating-T2 vs. other termination proving tools (see Fig. 6 for comparison to T2). Scatter plots are in log scale. Timeout=300s. NR="No Result", indicating failure of the tool.

	Term	Non-Term
Cooperating-T2	91.4%	96%
AProVE	73.5%	n.a.
KITTeL	73.1%	n.a.
T2	70.5%	99%
AProVE+Interproc	69.0%	n.a.
Terminator	66.0%	100%
Size-Change/MCNP	58.2%	n.a.
ARMC	51.5%	n.a.

Fig. 8. Evaluation overview

The results of our test runs are displayed in Figs. 6–8. Fig. 6 contains two plots which chart the difference between our new procedure and T2's previous procedure, in log scale. Plot **(a)** represents the results when applied to programs that terminate. Plot **(b)** contains the results from non-terminating benchmarks. Here both configurations of T2 use an approach similar to the approach used in TNT [24]. Our method from Fig. 5 has a fixed overhead, making non-terminating proofs for examples where the first counterexample already suffices to find a non-termination argument slower. Additionally, our method exposes a performance/non-termination bug in Z3 in a few cases, leading to some additional timeouts. In non-terminating examples with many other terminating loops, our program simplifications speed up the search for a non-termination proof. Fig. 7 compares our procedure to the other termination proving tools (to accommodate that not all tools support non-termination proofs, we only consider those examples that are not known to be non-terminating here). Fig. 8 gives the percentages of benchmarks proved (non-)terminating by the respective tools. The improvement for terminating benchmarks is dramatic: Cooperating-T2 times out or fails far less often than competing tools. On non-terminating benchmarks, the difference is small.

Discussion. Overall, the performance gains of our approach over previous techniques are dramatic. Our method does not just speed up termination proving, it makes a dramatic improvement in cases where previous tools time out or fail. Our experimental results also show how important supporting invariants are, *e.g.* in Cooperating-T2 vs. AProVE we see that many results cannot be obtained with the invariant **true**, even though AProVE also uses modern rank function synthesis algorithms (*e.g.* [2]). Furthermore, the result of AProVE+Interproc indicates that an eager search for invariants in a preprocessing step is not a suitable solution to this problem, as this leads to more timeouts. In-depth analysis shows that these are not only due to timeouts in the preprocessing tool Interproc, but that the wealth of generated invariants also slows down the later termination proof. As expected [16], the performance of ARMC and Terminator is worse than that of T2. Thus, since our approach improves dramatically over T2, it also represents an improvement over ARMC and Terminator.

5 Conclusion

One of the difficulties for reliable and scalable program termination provers is orchestrating the interplay between the reasoning about progress and the search for supporting invariants. In this paper we have developed a new method that facilitates cooperation between these two types of reasoning. Our representation

gives the underlying tools the whole picture of the current proof state, allowing both types of reasoning to contribute towards the greater goal and also to share their intermediate findings. As we have demonstrated experimentally, our approach leads to dramatic performance gains.

Future Work. We have focused on a method to improve performance of termination analysis for arithmetic programs. Our technique could be adapted for additional contexts. For example, a finite-state model checker could potentially make use of similar information when proving safety properties resulting from the liveness-to-safety reduction from Biere *et al.* [6]. The techniques developed here can possibly be adapted to proving termination of heap-based programs, perhaps by using shape analysis techniques in the safety subgraph to learn arithmetic invariants for the termination subgraph. Finally, we expect that the approach developed here adapts naturally to the problem of CTL and LTL model checking (*e.g.* via [13] and [14]), but we have not looked into this in detail yet.

Acknowledgments. We thank Christian von Essen, Jürgen Giesl, Heidy Khlaaf, Peter O'Hearn, Carsten Otto, and Abigail See for valuable discussions and the anonymous reviewers for helpful comments.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: WHALE: An interpolation-based algorithm for inter-procedural verification. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 39–55. Springer, Heidelberg (2012)
2. Alias, C., Darte, A., Feautrier, P., Gonnord, L.: Multi-dimensional rankings, program termination, and complexity bounds of flowchart programs. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 117–133. Springer, Heidelberg (2010)
3. Arts, T., Giesl, J.: Termination of term rewriting using dependency pairs. Theoretical Computer Science 236(1-2) (2000)
4. Ball, T., Rajamani, S.K.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
5. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. In: Proc. POPL 2007 (2007)
6. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. In: Proc. FMICS 2002 (2002)
7. Bradley, A.R., Manna, Z., Sipma, H.B.: Linear ranking with reachability. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 491–504. Springer, Heidelberg (2005)
8. Bradley, A.R., Manna, Z., Sipma, H.B.: The polyranking principle. In: Caires, L., Italiano, G.F., Monteiro, L., Palamidessi, C., Yung, M. (eds.) ICALP 2005. LNCS, vol. 3580, pp. 1349–1361. Springer, Heidelberg (2005)
9. Brockschmidt, M., Cook, B., Fuhs, C.: Better termination proving through cooperation. Technical Report AIB 2013-06, RWTH Aachen University, `http://aib.informatik.rwth-aachen.de`
10. Bruynooghe, M., Codish, M., Gallagher, J.P., Genaim, S., Vanhoof, W.: Termination analysis of logic programs through combination of type-based norms. ACM Trans. Program. Lang. Syst. 29(2) (2007)

11. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-Based Predicate Abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
12. Codish, M., Gonopolskiy, I., Ben-Amram, A.M., Fuhs, C., Giesl, J.: SAT-based termination analysis using monotonicity constraints over the integers. Theory and Practice of Logic Programming 11(4-5) (2011)
13. Cook, B., Koskinen, E.: Making prophecies with decision predicates. In: Proc. POPL 2011 (2011)
14. Cook, B., Koskinen, E., Vardi, M.: Temporal property verification as a program analysis task. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 333–348. Springer, Heidelberg (2011)
15. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: Proc. PLDI 2006 (2006)
16. Cook, B., See, A., Zuleger, F.: Ramsey vs. lexicographic termination proving. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 47–61. Springer, Heidelberg (2013)
17. Dershowitz, N.: Termination of rewriting. J. Symb. Comput. 3(1-2) (1987)
18. Falke, S., Kapur, D., Sinz, C.: Termination analysis of C programs using compiler intermediate languages. In: Proc. RTA 2011 (2011)
19. Floyd, R.W.: Assigning meaning to programs. In: Proc. of Symposia in Applied Mathematics. Mathematical Aspects of Computer Science. American Mathematical Society (1967)
20. Geser, A.: Relative Termination. PhD thesis, Universität Passau, Germany (1990)
21. Giesl, J., Schneider-Kamp, P., Thiemann, R.: aProVE 1.2: automatic termination proofs in the dependency pair framework. In: Furbach, U., Shankar, N. (eds.) IJCAR 2006. LNCS (LNAI), vol. 4130, pp. 281–286. Springer, Heidelberg (2006)
22. Giesl, J., Thiemann, R., Schneider-Kamp, P., Falke, S.: Mechanizing and improving dependency pairs. J. Autom. Reasoning 37(3), 155–203 (2006)
23. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: Proc. PLDI 2012 (2012)
24. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: Proc. POPL 2008 (2008)
25. Harris, W.R., Lal, A., Nori, A.V., Rajamani, S.K.: Alternation for termination. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 304–319. Springer, Heidelberg (2010)
26. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with BLAST. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 235–239. Springer, Heidelberg (2003)
27. Hirokawa, N., Middeldorp, A.: Automating the dependency pair method. Information and Computation 199(1,2) (2005)
28. Jeannet, B., Miné, A.: APRON: A library of numerical abstract domains for static analysis. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 661–667. Springer, Heidelberg (2009)
29. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: Proc. POPL 2001 (2001)
30. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic numeric abstractions for heap-manipulating programs. In: Proc. POPL 2010 (2010)
31. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
32. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: Joshi, R., Müller, P., Podelski, A. (eds.) VSTTE 2012. LNCS, vol. 7152, pp. 146–161. Springer, Heidelberg (2012)

33. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1) (2006)
34. Podelski, A., Rybalchenko, A.: ARMC: the logical choice for software model checking with abstraction refinement. In: Hanus, M. (ed.) PADL 2007. LNCS, vol. 4354, pp. 245–259. Springer, Heidelberg (2007)
35. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
36. Podelski, A., Rybalchenko, A.: Transition invariants. In: Proc. LICS 2004 (2004)
37. Press, W.H., Teukolsky, S.A., Vetterling, W.T., Flannery, B.P.: Numerical Recipes: The Art of Scientific Computing (1989)
38. Rybalchenko, A.: Private communication (2013)
39. Tsitovich, A., Sharygina, N., Wintersteiger, C.M., Kroening, D.: Loop summarization and termination analysis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 81–95. Springer, Heidelberg (2011)

Relative Equivalence in the Presence of Ambiguity

Oshri Adler, Cindy Eisner, and Tatyana Veksler

IBM Research - Haifa, Israel
{oshria,eisner,tatyana}@il.ibm.com

Abstract. We examine the problem of defining equivalence between two functions (pieces of code) that are intended to perform analogous tasks, but whose interfaces do not correspond in a straightforward way, even to the point of ambiguity. We formalize the notion of what equivalence means in such a case and show how to check it using constraints on a model checking problem. We show that the presence of constraints complicates the issue of predicate abstraction, and show that nevertheless we can use predicates no finer than those needed in the absence of constraints. Our solution is being used to verify the migration of tens of millions of lines of health insurance claims processing code from ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems (ICD), whose correspondence is complex and ambiguous in both directions. We present experimental results on 90,000 real life functions.

1 Introduction

Given two functions f and f' that are over the same domain, or in the terminology of hardware and software, have the same interface, it is easy to define equivalence between them, and to see what it means to check that equivalence. Also, if there is a one-to-one mapping between the domains of f and f', then the definition of equivalence and what it means to check it is trivial. In the real world, however, things are often not so neat and simple. Sometimes we have two functions that are intended to perform analogous tasks, but whose interfaces do not correspond in a straightforward way, even to the point of ambiguity. Given such a setting, what does it mean for f and f' to be equivalent and how can it be checked?

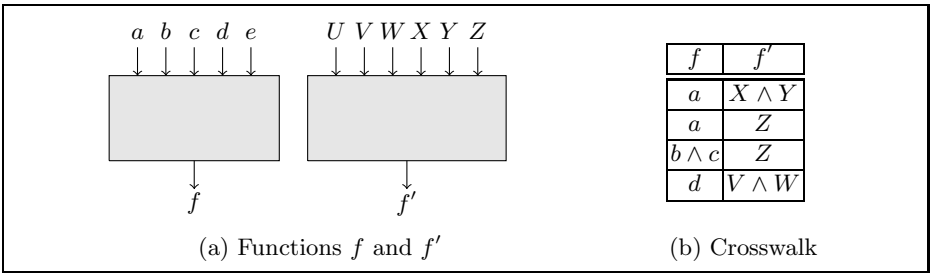
We encountered this interesting problem as part of an IBM engagement with NASCO®, an Atlanta, Georgia based company providing healthcare IT solutions to Blue Cross® and Blue Shield® (BCBS) Plans across the United States. The goal was to verify migration of insurance claims processing software, henceforth *benefit code*, from World Health Organization standard ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems (ICD) [1]. Correct migration of benefit code is of paramount importance to insurers, as benefit code directly affects the outflow of money.

Correspondence between ICD-9 and ICD-10 is given by a *schema crosswalk*, a table showing analogous elements of the interfaces. The correspondence is

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 430–446, 2013.

Table 1. Excerpt from an ICD-9/ICD-10 crosswalk

ICD-9		ICD-10	
041.81	Mycoplasma	$A49.3$	Mycoplasma infection, unspecified site
041.81	Mycoplasma	$B96.0$	Mycoplasma pneumoniae as the cause of diseases classified elsewhere
041.81 ∧ 466.0	Mycoplasma plus acute bronchitis	$J20.0$	Acute bronchitis due to mycoplasma pneumoniae
466.0	Acute bronchitis	$J20.9$	Acute bronchitis, unspecified



(a) Functions f and f'

f	f'
a	$X \wedge Y$
a	Z
$b \wedge c$	Z
d	$V \wedge W$

(b) Crosswalk

Fig. 1. A simple example

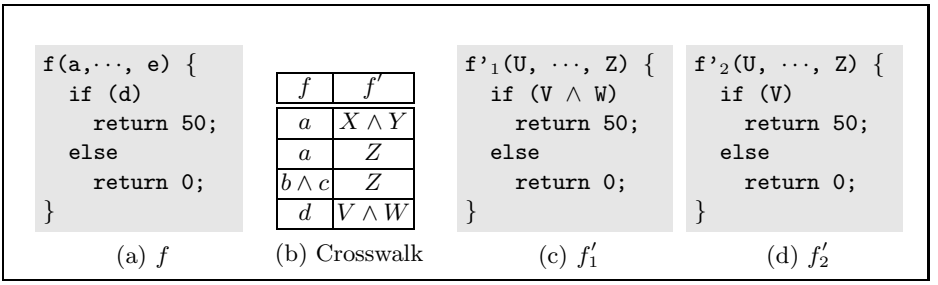
complex and ambiguous in both directions. For example, consider the excerpt shown in Table 1. ICD-9 diagnosis 041.81 corresponds to ICD-10 diagnosis $A49.3$ but also to $B96.0$, and if 041.81 appears in conjunction with 466.0, then together they correspond to diagnosis $J20.0$. But 466.0 by itself corresponds to $J20.9$.

Furthermore, not every ICD-9 code is expressible in ICD-10 and vice versa. For example, ICD-9 code $E927.0$ (Overexertion from sudden strenuous movement) has no comparable code in ICD-10, while ICD-10 code $T36.0X6$, (Underdosing of penicillins, initial encounter) has no comparable code in ICD-9.

The general problem is illustrated in Fig. 1: We are given two functions f and f' with completely different interfaces and a crosswalk describing how a single (Boolean) input of f is expressed in the interface of f' as the conjunction of one or more (Boolean) inputs of f', or vice versa. In this and all examples, we use lower case letters for inputs of f and upper case letters for inputs of f'.

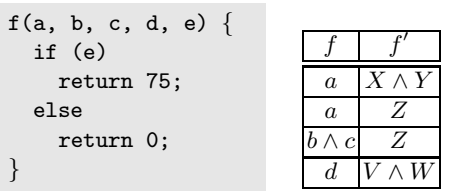
If an input is not expressible in the other interface, it is not mapped by the crosswalk. Input e of f and input U of f' are such inputs. Also, some inputs are expressible only in conjunction with others. For instance, $b \wedge c$ on the interface of f is expressible as Z in the interface of f', but there is no way to express b without c in the interface of f'. Finally, some inputs are expressible in more than one way – for instance, a on the interface of f and Z on the interface of f'.

Given such a crosswalk, our mission is to decide whether f and f' are equivalent relative to it. Intuitively, this means that they return the same value for analogous inputs. For example, for the f and f' shown in Fig. 1, we expect that f returns the same value on input d as an equivalent f' returns on input $V \wedge W$. But what about inputs not expressible in the other interface? Do we care what they return? We do. Such cases represent an exposure for the insurance company.



(a) f (b) Crosswalk (c) f'_1 (d) f'_2

Fig. 2. We want f to be equivalent to f'_1, but not to f'_2



(a) f (b) Crosswalk

Fig. 3. A fundamental mismatch

If a doctor diagnoses $V = 1$ and $W = 0$ in ICD-10, we can't know what she would have diagnosed in ICD-9. Explicitly paying a non-default amount for a claim not explicitly payable in the other interface introduces an element of uncertainty into the financial forecast. Thus we will require that equivalent functions treat such inputs in a default way – e.g., by falling into the "else" case.

Related Work. The term schema crosswalk is a term from database theory, and the question of relative equivalence can be asked about any pair of databases related by a crosswalk. However, we are unaware of related work; to the best of our knowledge the issues we explore here have not been explored in the context of databases, where the emphasis is usually on merging the contents of two databases rather than comparing code that interfaces with the databases as is.

2 Relative Equivalence

We now explore the issue of relative equivalence in more detail and define it precisely. We want to define that f and f' are equivalent if they each treat explicitly only inputs expressible in the other interface, and return the same value for analogous inputs. For example, we want to define that f and f'_1 of Fig. 2 are equivalent, and that f and f'_2 are not.

Consider now Fig. 3: f treats input e, not expressible in the other interface, in a non-default manner, thus we do not want it to be equivalent to any f'. We call such cases a *fundamental mismatch* between f and the crosswalk, because the inequivalence of f to f' is due to the crosswalk and not to the behavior of f'.

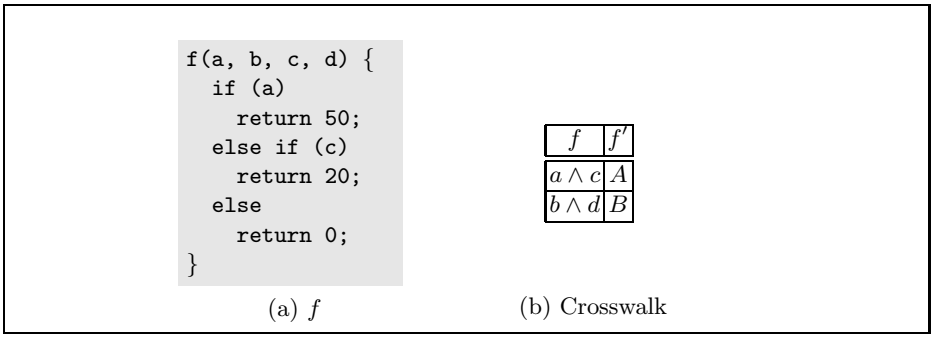


Fig. 4. Another fundamental mismatch

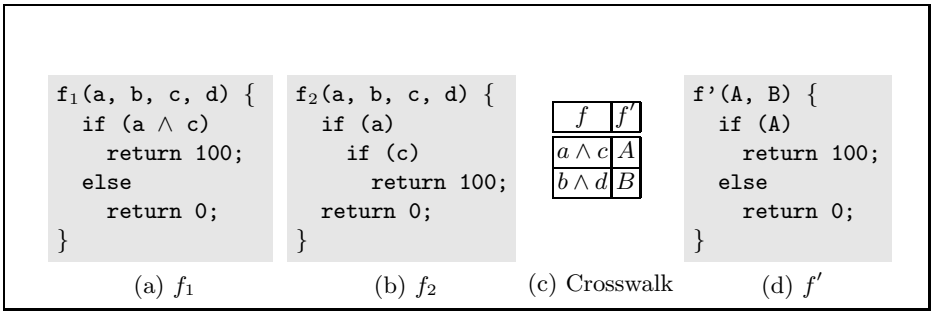


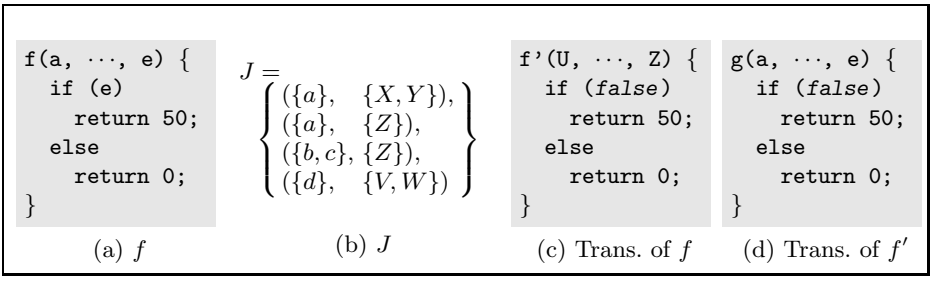
Fig. 5. f_1 and f_2 are equivalent, and we want both to be equivalent to f'

Figure 4 also shows a fundamental mismatch: f distinguishes between $a \wedge \neg c$, returning 50, and $\neg a \wedge c$, returning 20, but the interface of f' defined by the crosswalk cannot distinguish between them. It seems we should not allow the code to mention a without c or c without a, but this is going too far. The f_1 and f_2 of Fig. 5 are equivalent, so if f_1 is equivalent to f' (and intuitively, it is), then we want also that f_2 is equivalent to f'. Thus we must allow inputs not expressible in the other interface to stand alone, depending on the context.

That is, we want a semantic, not a syntactic, definition of equivalence, so we can identify equivalence between functions with differing control flows. Thus we will work over equivalence classes of input vectors to f and f', where an equivalence class is a set of input vectors for which the function returns the same value, and an input vector is a valuation of the individual inputs. For example, f_1 of Fig. 5 has two equivalence classes. One returns 100 and consists of vectors in which $a = c = 1$; the other returns 0, and consists of all other input vectors.

Definition 1 (Equivalence class). *An equivalence class of a function f is a maximal set of input vectors V such that for all $v_1, v_2 \in V$: $f(v_1) = f(v_2)$.*

We are now ready to define relative equivalence. Let P and P' be disjoint sets of atomic propositions. P and P' represent the inputs of f and f', respectively. The crosswalk is represented by the relation J, and we use $\varphi(v)$ and $\varphi(v')$ to move from elements of J back to conjunctions as used in the crosswalk.



(a) f (b) J (c) Trans. of f (d) Trans. of f'

Fig. 6. Translation of a fundamental mismatch

Definition 2 (Justified by the crosswalk (J)). *$J \subseteq 2^P \times 2^{P'}$ is a relation such that for every $(v, v') \in J$, either v or v' is a singleton.*

Definition 3 (Conjunctive formula ($\varphi(v), \varphi(v')$)). *Let $v \subseteq P$ and $v' \subseteq P'$. Then $\varphi(v)$ denotes the formula $\bigwedge_{p \in v} p$ and $\varphi(v')$ denotes the formula $\bigwedge_{p' \in v'} p'$.*

For example, the crosswalk of Fig. 5 is formalized as $J = \{(\{a, c\}, \{A\}), (\{b, d\}, \{B\})\}$ and we have that $\varphi(\{a, c\}) = a \wedge c$.

We now define translations between propositional formulas over P and P'.

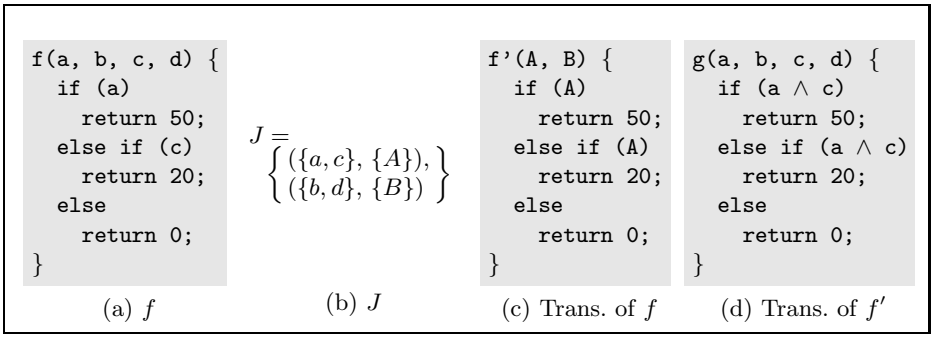
Definition 4 (Translation ($T(\psi), T'(\psi')$)). *Let Φ and Φ' be the set of proposition formulas over P and P', respectively. Let $p \in P$, $p' \in P'$ and $\psi, \psi_1, \psi_2 \in \Phi$, $\psi', \psi'_1, \psi'_2 \in \Phi'$. The functions $T : \Phi \mapsto \Phi'$ and $T' : \Phi' \mapsto \Phi$ are defined as follows:*

- $T(p) = \bigvee_{\{v' \mid (v,v') \in J \text{ and } p \in v\}} \varphi(v')$
- $T(\psi_1 \wedge \psi_2) = T(\psi_1) \wedge T(\psi_2)$
- $T(\psi_1 \vee \psi_2) = T(\psi_1) \vee T(\psi_2)$
- $T(\neg\psi) = \neg T(\psi)$

- $T'(p') = \bigvee_{\{v \mid (v,v') \in J \text{ and } p' \in v'\}} \varphi(v)$
- $T'(\psi'_1 \wedge \psi'_2) = T'(\psi'_1) \wedge T'(\psi'_2)$
- $T'(\psi'_1 \vee \psi'_2) = T'(\psi'_1) \vee T'(\psi'_2)$
- $T'(\neg\psi') = \neg T'(\psi')$

In the sequel, f and f' will be defined as *relatively equivalent* if the formulas characterizing their equivalence classes translate into each other. Before proceeding to the formal definition, let's take a look at how we can translate functions by translating the conditional expressions that form the basis of the equivalence classes, and what happens when we try to translate propositions not expressible in the other interface. Consider Fig. 5: $T(a) = T(c) = A$, and $T'(A) = a \wedge c$. Therefore the "if" statement of f_1 and both "if" statements of f_2 translate into "`if (A)`". Also, the "if" statement of f' translates into "`if(a ∧ c)`". Thus we can use T and T' to show that f_1 and f_2 are both equivalent to f'.

When we attempt to do the same with f's that are fundamental mismatches, we will get that translating forwards and back will not get us to where we started. Consider for example Fig. 6. We have that f translates into f', because there are no lines of the crosswalk containing e, so $T(e)$ is the empty disjunction *false*. Then, $\mathit{false} = p \wedge \neg p$ for some p, so we get that $T'(\mathit{false}) = \mathit{false}$ and translating f' back into the other interface gives us the g of Fig. 6d. In particular, f' does not



(a) f (b) J (c) Trans. of f (d) Trans. of f'

Fig. 7. Translation of another fundamental mismatch

translate back to f. The same thing will happen with the fundamental mismatch we saw in Fig. 4 – its translation forward and back is shown in Fig. 7.

Formally, an input vector of a function is a *valuation* v of a set Q (e.g., P or P'), associating each proposition with a value **T** or **F**. Note that equivalently, a valuation v can be associated with the subset $\{q \in Q \mid q \text{ has the value } \mathbf{T} \text{ in } v\}$. We abuse notation and use valuations of Q and subsets of Q (elements of 2^Q) interchangeably. For a valuation v and a propositional formula ψ, we use $v \models \psi$ to denote that ψ holds on valuation v and we use $[\psi]$ to denote the set of valuations on which ψ holds. We use $\psi_1 \equiv \psi_2$ to denote that $[\psi_1] = [\psi_2]$.

Recall that an equivalence class is just a set of valuations and thus can be denoted by $[\psi]$ for some formula ψ. For example, the equivalence classes of f of Fig. 7a are $[a]$, $[\neg a \wedge c]$ and $[\neg a \wedge \neg c]$. For an equivalence class $[\psi_i]$ of a function f, let $f([\psi_i])$ denote the value returned by f for every element of $[\psi_i]$. For simplicity we assume that there is a total order on the values returned by f and f'; if there is not, choose an order arbitrarily. Thus we can assume wlog that equivalence classes are ordered such that $f([\psi_i]) < f([\psi_{i+1}])$. We now define relative equivalence based on T and T' of Definition 4 as follows:

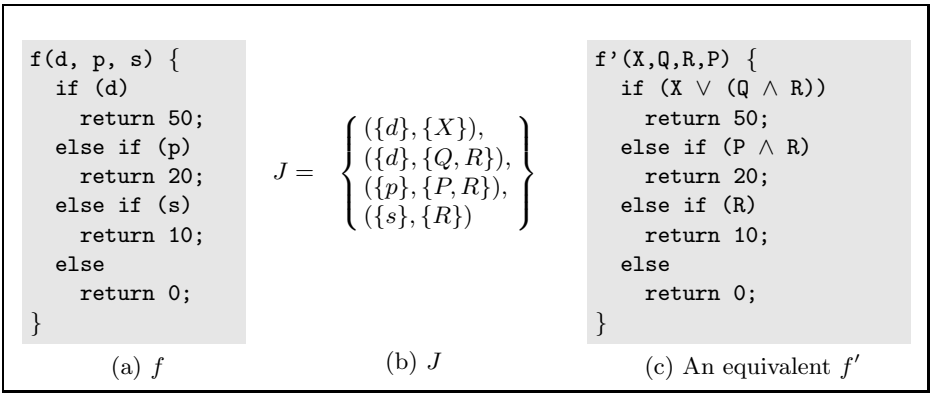
Definition 5 (Relative equivalence). *Let $f : 2^P \mapsto \mathcal{R}$ and $f' : 2^{P'} \mapsto \mathcal{R}$ for some range $\mathcal{R}$, and let $[\psi_i]$ for $i \leq n$ be the equivalence classes of f and $[\psi'_j]$ for $j \leq n'$ be the equivalence classes of f'. Then f is equivalent to f' relative to J, denoted $f \sim f'$, if:*

$$n = n' \text{ and for every } i \leq n : f([\psi_i]) = f'([\psi'_i]) \text{ and } \psi_i \equiv T'(\psi'_i) \text{ and } \psi'_i \equiv T(\psi_i)$$

The definition of T and T', and thus of relative equivalence, is based on syntax, but these are semantic notions, as stated by the following proposition.

Proposition 1. *Let $\psi_1, \psi_2 \in \Phi$ such that $\psi_1 \equiv \psi_2$ and $\psi'_1, \psi'_2 \in \Phi'$ such that $\psi'_1 \equiv \psi'_2$. Then $T(\psi_1) \equiv T(\psi_2)$ and $T'(\psi'_1) \equiv T'(\psi'_2)$.*

Consider now Fig. 8. The equivalence classes of f are given by $\psi_4 = d$, $\psi_3 = \neg d \wedge p$, $\psi_2 = \neg d \wedge \neg p \wedge s$ and $\psi_1 = \neg d \wedge \neg p \wedge \neg s$ (recall that we order equivalence classes according to their return value) and those of f' are given by $\psi'_4 = X \vee (Q \wedge R)$,



(a) f (b) J (c) An equivalent f'

Fig. 8. An equivalent f and f', this time formally

$\psi'_3 = \neg(X \vee (Q \wedge R)) \wedge (P \wedge R) \equiv \neg X \wedge \neg Q \wedge P \wedge R$, $\psi'_2 = \neg(X \vee (Q \wedge R)) \wedge \neg(P \wedge R) \wedge R \equiv \neg X \wedge \neg Q \wedge \neg P \wedge R$ and $\psi'_1 = \neg(X \vee (Q \wedge R)) \wedge \neg(P \wedge R) \wedge \neg R \equiv \neg X \wedge \neg R$. Also, from J we have that $T(d) = X \vee (Q \wedge R)$, $T(p) = P \wedge R$, $T(s) = R$, $T'(X) = d$, $T'(Q) = d$, $T'(P) = p$ and $T'(R) = d \vee p \vee s$. Substituting gives $T(\psi_i) = \psi'_i$ and $T'(\psi'_i) = \psi_i$ for $1 \leq i \leq 4$, thus $f \sim f'$.

Now consider functions f and f' that switch the first and third conditions in Figs. 8a and 8c. Doing so would change the equivalence classes. In particular we would have that $\psi_4 = s$ and $\psi'_4 = R$, but $T'(R) = d \vee p \vee s \not\equiv s$, thus such an f and f' would not be relatively equivalent.

3 Checking Relative Equivalence

Having defined relative equivalence, how can it be checked? One way would be to calculate equivalence classes and check whether they are in the correct relation. However, real life benefit code functions calculate tens of outputs, so doing so would require multiple semantic analyses. Thus we prefer a way that avoids calculating equivalence classes and instead uses a single run of a model checker. We are looking for a relation $R \subseteq 2^P \times 2^{P'}$, such that $f \sim f'$ can be checked by checking whether $f(v) = f'(v')$ for every $(v, v') \in R$.

In the sequel, we represent an input vector by a set, where the elements of the set are the inputs that have the value 1. For example, for the f of Fig. 8a, $\{d, s\}$ represents the input vector in which $d = s = 1$ and $p = 0$. Now, it seems that checking equivalence of f and f' relative to a crosswalk should entail checking that f and f' return the same value for corresponding input vectors, where by correspond we mean that they are analogous according to the crosswalk. For example, consider the J of Fig. 9. Input vector $\{a\}$ of f corresponds to input vector $\{X, Y\}$ of f', input vector $\{b\}$ of f corresponds to input vector $\{Z\}$ of f', and input vector $\{a, b\}$ of f corresponds to input vector $\{X, Y, Z\}$ of f'.

Is it sufficient to check all pairs of corresponding input vectors? No. Doing so will result in a verdict of "equivalent" for the f and f'_1 shown in Fig. 9, even

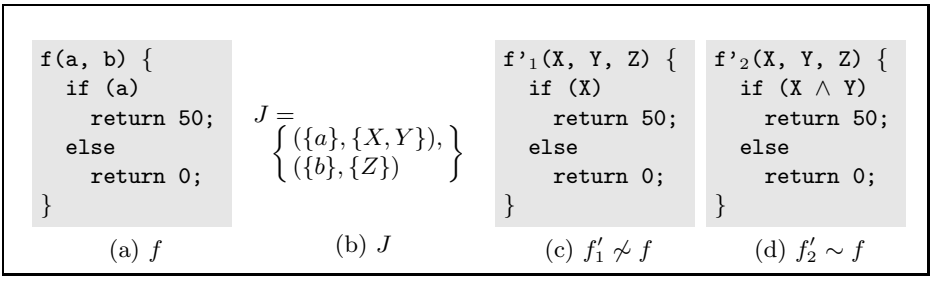


Fig. 9. Checking all pairs of corresponding input vectors is insufficient

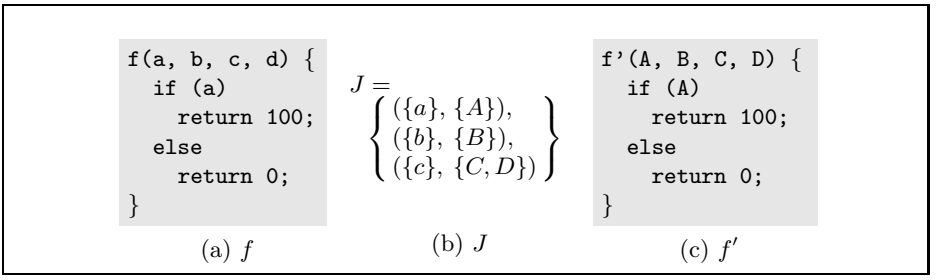


Fig. 10. Input vector $\{A, B, C\}$ of f behaves in a non-default way

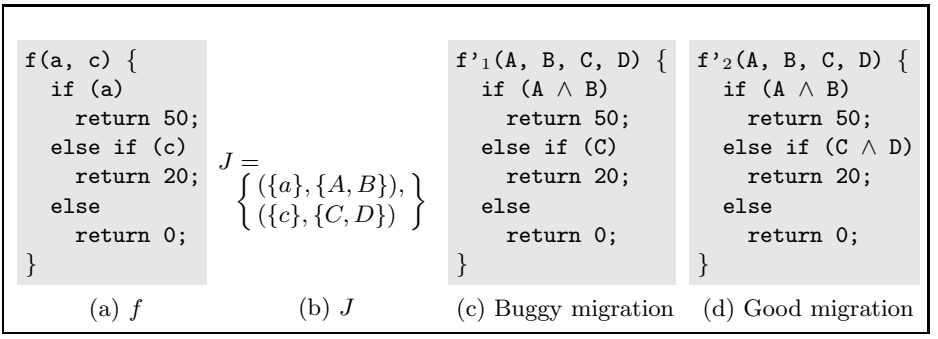


Fig. 11. A good and a buggy migration

though f is equivalent to f'_2 but not to f'_1. The only input vectors that can distinguish between f'_1 and f'_2 are ones in which X holds but Y does not, but such vectors have no corresponding vector in the other interface. It seems we can deal with this by requiring a "default" behavior from such input vectors, where by default we mean that the calculation falls into some "else" case. However, consider the example of Fig. 10. Input vector $\{A, B, C\}$ of f' has no equivalent in f, yet behaves in a non-default way. It makes no sense to require that $\{A, B, C\}$ falls into the "else" case, since f and f' are clearly equivalent as written.

So how should we pair a vector with no corresponding vector in the other interface? Consider Fig. 11, in which f has been migrated twice to a new interface.

Input vector $\{C\}$ has no corresponding input vector according to J, because C does not "stand alone" in any element of J. Thus we expect that in a correct migration, C cannot influence the returned value without D, so we expect that a correct migration f' returns the same value for $\{C\}$ that it returns on $\emptyset$, and in general that for any input vector v' of f' such that $C \in v'$ but $D \notin v'$, we have that $f'(v') = f'(v' \setminus \{C\})$. We call C in vectors $\{C\}$ and $\{A, B, C\}$ an *orphan* input, because the vector does not contain enough other inputs to complete a line in the crosswalk. The *remainder* of an input vector consists of all its orphans:

Definition 6 (Remainder ($r(v)$, $r'(v')$)). *$r : 2^P \mapsto 2^P$ and $r' : 2^{P'} \mapsto 2^{P'}$ are defined as follows:*

$$\begin{aligned} r(v) &= \{p \in v \mid \nexists w \subseteq v,\ w' \subseteq P' \ \ s.t.\ (\{p\} \cup w,\ w') \in J\} \\ r'(v') &= \{p' \in v' \mid \nexists w \subseteq P,\ w' \subseteq v' \ \ s.t.\ (w, \{p'\} \cup w') \in J\} \end{aligned}$$

For example, using the J of Fig. 11, we have that $r'(\{A, B, C\}) = \{C\}$.

Using the notion of remainder, we define the relation R as follows:

Definition 7 (Relevant input pairs (R)). *The set of relevant input pairs is given by $R \subseteq 2^P \times 2^{P'}$ defined as follows:*

$$\begin{aligned} R = \{(v, v') \mid\ & v \in 2^P \ \ \textit{and}\ v' \in 2^{P'} \ \ \textit{and}\ \exists v_1, v_2, \cdots v_k, v'_1, v'_2, \cdots v'_k \ \ s.t. \\ & v = v_1 \cup v_2 \cup \cdots \cup v_k \ \ \textit{and}\ v' = v'_1 \cup v'_2 \cup \cdots \cup v'_k \ \ \textit{and} \\ & v_k = r(v) \ \ \textit{and}\ v'_k = r'(v') \ \ \textit{and}\ \forall i < k : (v_i, v'_i) \in J\} \end{aligned}$$

For example, using the J of Fig. 11, we have that $(\{a, c\}, \{A, B, C, D\}) \in R$ because $\{a, c\} = v_1 \cup v_2 \cup v_3$ and $\{A, B, C, D\} = v'_1 \cup v'_2 \cup v'_3$ for $v_1 = \{a\}$, $v_2 = \{c\}$, $v_3 = \emptyset$, $v'_1 = \{A, B\}$, $v'_2 = \{C, D\}$ and $v'_3 = \emptyset$, and we have that $(v_1, v'_1), (v_2, v'_2) \in J$, $v_3 = r(\{a, c\})$, and $v'_3 = r'(\{A, B, C, D\})$. Also, $(\emptyset, \{C\})$, a vector pair that finds the migration bug, is in R, because $\emptyset = v_1$ and $\{C\} = v'_1$ for $v_1 = \emptyset$ and $v'_1 = \{C\}$ and $v_1 = r(\emptyset)$ and $v'_1 = r'(\{C\})$.

R includes every pair of corresponding vectors (and then the remainders are empty) and also pairs vectors without a corresponding vector in a way that expects that the remainders don't influence the function. Intuitively, checking that f and f' return the same value for every pair in R should be a way to show that f and f' are relatively equivalent. The following theorem confirms this.

Theorem 1 (Checking relative equivalence)

$$f \sim f' \ \ \textit{iff}\ \forall (v, v') \in R : f(v) = f'(v')$$

The proof of the $\Longrightarrow$ direction is based on the following lemma.

Lemma 1 (Relating R and T, T'). *Let $(v, v') \in R$ and let $\pi \in \Phi$ and $\pi' \in \Phi'$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$. Then $v \models \pi \iff v' \models \pi'$.*

To prove the $\Longleftarrow$ direction, we observe that every equivalence class containing $v \in 2^P$ must contain as well all valuations in 2^P that are transitively related to v by R, and similarly for $v' \in 2^{P'}$. We define:

Definition 8 (Expected same-behavior valuations of (v, v') ($E(v, v')$). *Let $(v, v') \in R$. Then $E(v, v')$ is the subset of $2^P \cup 2^{P'}$ defined inductively as follows:*

- $v, v' \in E(v, v')$.
- *If $w \in E(v, v')$ and $(w, w') \in R$, then $w' \in E(v, v')$*
- *If $w' \in E(v, v')$ and $(w, w') \in R$, then $w \in E(v, v')$*

The challenge is to show that each $E(v, v')$ can be represented as $[\pi] \cup [\pi']$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$, for some π, π', as stated by the following lemma:

Lemma 2 (Expressing $E(v, v')$). *Let $(v, v') \in R$. Then $\exists \pi, \pi'$ such that $\pi \equiv T'(\pi')$ and $\pi' \equiv T(\pi)$ and $E(v, v') = [\pi] \cup [\pi']$.*

The proof of Lemma 2 is based on showing that $E(v, v')$ "has no holes", that is, that if u and z are in $E(v, v')$, then every w such that $u \subseteq w \subseteq z$ is also in $E(v, v')$. This allows us to define a partial order on the various $E(v, v')$, and we form the necessary formulas starting from the single maximal element in the partial order, $E(P, P')$. Taking the disjunction of $\varphi(u)$ for all minimal elements $u \in 2^P$ in $E(v, v')$ gives us a formula characterizing every valuation $x \in 2^P$ that is in $E(v, v')$ or in some $E(w, w') \succ E(v, v')$. From there it is a simple matter to remove the valuations that are too big, by conjuncting with the negation of the formulas formed previously for larger $E(w, w')$'s.

4 Model Checking Setup

We have a theory of relative equivalence and a set of input pairs sufficient to check it. Using these, we set up our model checking problem as follows. We compile a pair of benefit code functions into an SMV [7] model constructed in a straightforward manner, similar to the method described in [5,6]. A dedicated state variable keeps track of the control flow, whose behavior may depend on the value of other state variables. Each input is allocated a state variable, which wakes up in a non-deterministic state and keeps its value throughout the run. Each variable of the original code is also allocated a state variable, and is assigned a value when the control flow reaches a relevant line.

The actual benefit code language is proprietary. While it does not contain loops, it does contain some constructs that are quite tricky to model. Due to space constraints, the details are beyond the scope of this paper. In Fig. 12a we show an example that uses pseudo-code based on the syntax of C, similarly to previous examples. The f and f' shown in Fig. 12a might compile to the model M_c shown in Fig. 12b. The behaviors of variables line1 and line2 represent the control flow of f and f', respectively, and the behaviors of variables pay1 and pay2 represent the behavior of variables pay in f and f', respectively.

It remains to constrain the inputs to pairs in R and to check that `pay1 = pay2` whenever both computations have ended (`line1 = 4` and `line2 = 4`).

For brevity, declarations and initializations are not shown in the SMV *model.*

```
  f(a, b) {
1   if (a ∧ b)
2     pay = 50;
    else
3     pay = 0;
4 }
```

```
  f'(V, W) {
1   if (V ∨ W)
2     pay = 50;
    else
3     pay = 0;
4 }
```

(a) f and f'

```
assign next(line1) := case
   line1=1:  if (a ∧ b) then 2 else 3 endif;
   else   :  4;
esac;

assign next(line2) := case
   line2=1:  if (V ∨ W) then 2 else 3 endif;
   else   :  4;
esac;

assign next(pay1) := case
   line1 = 2:  50;
   line1 = 3:  0;
   else:       pay1;
esac;

assign next(pay2) := case
   line2 = 2:  50;
   line2 = 3:  0;
   else:       pay2;
esac;
```

(b) The (concrete) SMV model M_c

Fig. 12. Compiling f and f' to an SMV model

For $p \in P$, let $J^p = \{(u, u') \in J \mid p \in u\}$ and similarly for $p' \in P'$ let $J^{p'} = \{(u, u') \in J \mid p' \in u'\}$. We constrain each element $p \in P$ and $p' \in P'$ as follows:

$$p \leftrightarrow \Big(\Big(\bigvee_{(u,u')\in J^p} (\varphi(u) \wedge \varphi(u'))\Big) \bigvee \Big(p \wedge \bigwedge_{(u,u')\in J^p} \neg\varphi(u)\Big)\Big) \tag{1}$$

$$p' \leftrightarrow \Big(\Big(\bigvee_{(u,u')\in J^{p'}} (\varphi(u) \wedge \varphi(u'))\Big) \bigvee \Big(p' \wedge \bigwedge_{(u,u')\in J^{p'}} \neg\varphi(u')\Big)\Big) \tag{2}$$

Definition 9 (Concrete constraint (C_c)). *Let C_c be the conjunction of Equations (1) and (2) for each $p \in P$ and $p' \in P'$.*

Constraining our model checking problem using C_c allows us to check relative equivalence, as stated by the following theorem.

Theorem 2 (Using the concrete constraint)

$$f \sim f' \text{ iff } \forall (v, v') \text{ s.t. } (v \cup v') \models C_c : f(v) = f'(v')$$

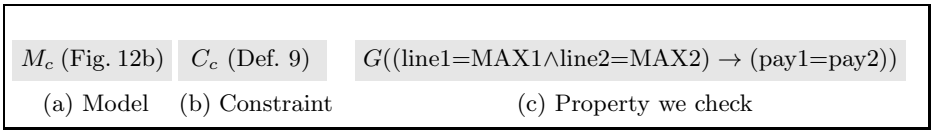


Fig. 13. The complete (concrete) model checking problem

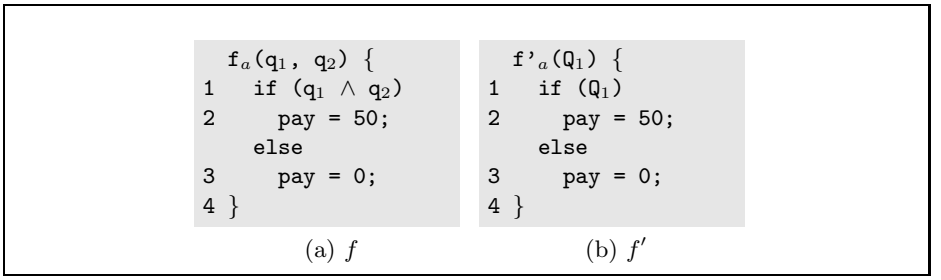


Fig. 14. Abstract versions of the f and f' shown in Fig. 12

For example, let $J = \{(\{a\}, \{A\}), (\{b\}, \{B, C\})\}$, then our constraint would be:

$$\begin{array}{llll} (a & \leftrightarrow ((a \wedge A) & \vee (a \wedge \neg a))) & \wedge \\ (b & \leftrightarrow ((b \wedge B \wedge C) & \vee (b \wedge \neg b))) & \wedge \\ (A & \leftrightarrow ((a \wedge A) & \vee (A \wedge \neg A))) & \wedge \\ (B & \leftrightarrow ((b \wedge B \wedge C) & \vee (B \wedge \neg(B \wedge C)))) & \wedge \\ (C & \leftrightarrow ((b \wedge B \wedge C) & \vee (C \wedge \neg(B \wedge C)))) & \end{array} \tag{3}$$

giving

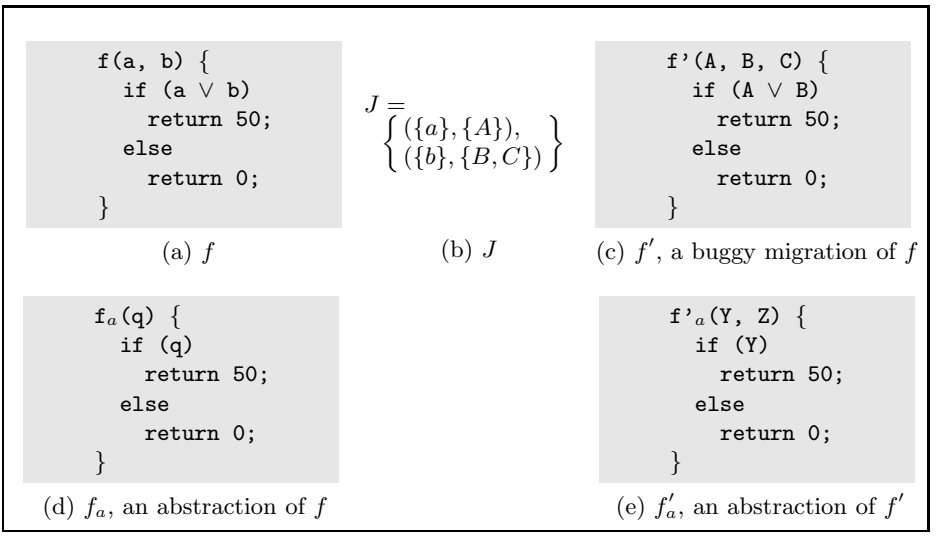
$$R = \left\{ \begin{array}{l} (\emptyset, \emptyset), (\emptyset, \{B\}), (\emptyset, \{C\}), (\{a\}, \{A\}), (\{a\}, \{A, B\}), \\ (\{a\}, \{A, C\}), (\{b\}, \{B, C\}), (\{a, b\}, \{A, B, C\}) \end{array} \right\} \tag{4}$$

Thus our complete (concrete) model checking problem, shown in Fig. 13, consists of the model M_c, the constraint C_c, and the property shown in Fig. 13c.

4.1 Complications Arising from Predicate Abstraction

A single line of benefit code can access files called *tables* representing large disjunctions of inputs, often consisting of hundreds of disjuncts each, so checking even a small function might involve thousands of state variables. Thus to avoid the size problem we use abstract versions of f and f', built by using predicates to represent disjunctions. We use the coarsest such abstraction that does not lose precision, thus our abstractions are *exact* in the sense of [3,4]. For example, the abstractions of the f and f' of Fig. 12 are shown in Fig. 14. Recall that we allocate predicates to disjunctions but not to conjunctions, thus f_a uses two predicates while f'_a uses only one. We define:

Definition 10 (Represents). *Let S be a set of atomic propositions, let $\{S_1, S_2, \cdots, S_\ell\}$ be a partition of S and for every S_i, let q_i abstract $\bigvee_{p \in S_i} p$. Then q_i represents s if $s \in S_i$.*



(a) f (b) J (c) f', a buggy migration of f

(d) f_a, an abstraction of f (e) f'_a, an abstraction of f'

Fig. 15. A buggy migration and its predicate abstraction

It is easy to see that if we partition P and P' carefully, we can get that the model M_a built from f_a and f'_a is bisimulation equivalent to the model M_c built from f and f'. However, taking our constraints into consideration, the abstraction seems to break down. For example, let J_a be obtained from J by replacing every element $p \in P \cup P'$ with the predicate q that represents it, and then constrain every $q \in A$ using a version of Equation (1) that uses J_a instead of J, and similarly for every $q' \in A'$. The constraints may make finer distinctions than those made by our predicates, thus using this abstraction we miss bugs.

For example, consider the buggy migration and its predicate abstraction shown in Fig. 15. Using predicates q abstracting $a \vee b$, Y abstracting $A \vee B$ and Z abstracting C, we get $J_a = \{(\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\}$, which gives:

$$\begin{array}{l} (q \leftrightarrow ((q \wedge Y) \vee (q \wedge Y \wedge Z) \vee (q \wedge \neg q))) \qquad\qquad\qquad\quad \wedge \\ (Y \leftrightarrow ((q \wedge Y) \vee (q \wedge Y \wedge Z) \vee (Y \wedge \neg Y \wedge \neg(Y \wedge Z)))) \wedge \\ (Z \leftrightarrow ((q \wedge Y \wedge Z) \qquad\qquad\quad \vee (Z \wedge \neg(Y \wedge Z)))) \end{array} \tag{5}$$

representing the following (bad) abstract R, call it R_b:

$$R_b = \{(\emptyset, \emptyset), (\emptyset, \{Z\}), (\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\} \tag{6}$$

R_b is bad because f_a and f'_a return the same value for every input pair in R_b, thus we have missed the migration bug.

One solution would be to build finer predicates that take the concrete constraints into consideration, but given how our constraints are built, with every atomic proposition on one side of an $\leftrightarrow$, that would seem to leave us with no abstraction at all. Happily, we can avoid building finer predicates than the coarsest

required for bisimulation – this makes intuitive sense, since if f cannot distinguish between input vector v_1 and v_2, there cannot be any reason to check both. What we want is to check every pair in R_a, obtained from R by replacing every $p \in P$ and $p' \in P'$ with the predicate that represents it. All that remains is to constrain the abstract inputs to pairs in R_a.

We define our abstract constraint C_a as follows:

Definition 11 (Abstract constraint (C_a)). *Let $\{P_1, P_2, \cdots, P_\ell\}$ be a partition of P represented by predicates $\{q_1, q_2, \cdots q_\ell\}$ and let $\{P'_1, P'_2, \cdots P'_{\ell'}\}$ be a partition of P' represented by predicates $\{q'_1, q'_2, \cdots q'_{\ell'}\}$.*

$$C_a = \exists_{p \in P} \exists_{p' \in P'} \left(\bigwedge_{i=1}^{\ell} \Big(q_i \leftrightarrow \bigvee_{p \in P_i} p \Big) \ \wedge \ \bigwedge_{j=1}^{\ell'} \Big(q'_j \leftrightarrow \bigvee_{p' \in P'_j} p' \Big) \ \wedge \ C_c \right)$$

Using C_a, built at compile time, allows us to check that f and f' are relatively equivalent by comparing f_a and f'_a, as stated by the following theorem.

Theorem 3 (Using the abstract constraint). *Let f_a be an abstraction of f and let f'_a be an abstraction of f'. Then*

$$f \sim f' \text{ iff } \forall (x, x') \text{ s.t. } (x \cup x') \models C_a : f_a(x) = f'_a(x')$$

For example, using our predicates q abstracting $a \vee b$, Y abstracting $A \vee B$ and Z abstracting C, we get the following abstraction of the R from Equation (4):

$$R_a = \{(\emptyset, \emptyset), (\emptyset, \{Y\}), (\emptyset, \{Z\}), (\{q\}, \{Y\}), (\{q\}, \{Y, Z\})\} \tag{7}$$

Then the migration bug of Fig. 15 will be found by the abstract pair $(\emptyset, \{Y\})$, representing the concrete pair $(\emptyset, \{B\})$.

Our complete abstract model checking problem, then, consists of the abstract model M_a, the abstract constraint C_a, and the property shown in Fig. 13c.

5 Experimental Results

We implemented our method in a tool to formally verify migration of benefit code from ICD-9 to ICD-10 as part of an IBM engagement with NASCO. The result is being used to verify migration of tens of millions of lines of benefit code, consisting of millions of relatively small functions. As part of our testing process we gathered statistics on a subset of the real code, consisting of some 90,000 functions, comparing them to an ad hoc migration developed specially for testing purposes. We used GEM files published by the Centers for Medicare and Medicaid Services [2], in which J consists of approximately 175,000 pairs, mapping some 17,000 ICD-9 diagnosis and procedure codes to some 141,000 ICD-10 codes. In this section we present some practical details regarding the implementation, and information on the performance of our tool on its realistic test base.

As we have seen, some functions are fundamental mismatches with respect to J, thus not migratable, and so correct-by-construction migration is not possible

Table 2. Size, compile and run time (in seconds)

Plan	# f's	Lines of Code				Compile Time				Run Time			
		Total	Ave	Med	Max	Total	Ave	Med	Max	Total	Ave	Med	Max
A	149	4241	28.5	22	160	56	0.4	0.1	9.9	86	0.6	0.5	11.3
B	13,673	344,184	25.2	21	632	21,521	1.6	0.2	11.4	158,197	11.6	6.0	445.3
C	20,395	538,281	26.4	26	632	38,115	1.9	0.2	12.4	128,421	6.3	5.5	459.2
D	4,727	135,746	28.7	23	428	1,966	0.4	0.5	12.2	2,187	0.5	0.5	509.5
E	49,523	395,781	8.0	7	232	7,497	0.2	0.1	11.6	44,825	0.9	0.6	999.0
F	1,607	77,347	48.1	32	330	3,655	2.3	0.3	11.2	156,049	97.1	1.5	965.3

Table 3. Comparison results

Plan	Count			%		
	Migration Correct	Fundamental Mismatch	Other Mismatch	Migration Correct	Fundamental Mismatch	Other Mismatch
A	138	11	0	92.6	7.4	0.0
B	13,673	0	0	100.0	0.0	0.0
C	13,141	7,200	54	64.4	35.3	0.3
D	4,584	139	4	97.0	2.9	0.1
E	49,248	169	106	99.4	0.3	0.2
F	1,400	175	32	87.1	10.9	2.0

without a check for migratability. Even for migratable functions, the process is not so simple: Not every expression is expressible in the proprietary language used in the benefit code, so some expressions need to be split into sequential or nested conditional statements in such a way that the migrated function f' might be syntactically far from the original function f; in particular, the control flow of f and f' may be quite different. For these and other reasons, the migration process consists of two steps. First a heuristic migration is performed that produces correct results in most but not all cases, then a verification step checks if the migration is correct. If it is not, a counterexample is produced and the migration is fixed manually and re-verified.

Our experiment consisted of applying our method to 90,000 real benefit code functions belonging to six different insurance plans, comparing them to an ad hoc migration developed specially for testing purposes. Our industrial strength model checker RuleBase PE [8], designed for very large model checking problems, incurs unnecessary overhead when applied to small problems. Instead, we run directly on Discovery, RuleBase PE's BDD-based model checking engine. Table 2 shows, for each plan, the total, average, median and maximum lines of code per function and compile and run time in seconds on a 2×2.4 GHz Intel Xeon processor with 2 GB RAM running Red Hat 5.6. For testing purposes, we timed out at 1,000 seconds run time. Out of just over 90,000 test cases, 140 timed out and are not included in the numbers shown in Table 2.

Table 3 shows the model checking results. In most cases, our ad hoc migration was correct. Some cases were identified as suspected fundamental

mismatches at compile time and those that were confirmed at run time are listed in the column labeled "Fundamental Mismatch". We find suspected fundamental mismatches at compile time when we discover during predicate abstraction that some atomic proposition should be used by f but is not. For example, if $(\{a, b\}, \{A\}) \in J$ and there is no other $(v, v') \in J$ such that $a \in v$, then an f that uses a but not b cannot be correctly migrated unless the use of a is in dead code. Run time distinguishes between real and spurious suspected fundamental mismatches. Note that the 7,200 fundamental mismatches found for Plan C most likely result from a smaller number of errors in some table used by multiple functions.

The other mismatches shown in Table 3 represent either fundamental mismatches not identified at compile time (e.g., in the above example, if both a and b are used in the code but in the wrong context) or bugs in our ad hoc migration.

6 Conclusion

We have formalized the notion of relative equivalence and characterized the set of cases R sufficient to check it. We have shown how predicate abstraction interacts with constraint generation and presented a solution that avoids overrefinement. We have implemented our solution in a tool currently being used to migrate tens of millions of lines of insurance claims processing code from from ICD-9 to ICD-10, two versions of the International Statistical Classification of Diseases and Related Health Problems. We have presented experimental results for the migration of 90,000 real functions from this code, using a crosswalk consisting of approximately 175,000 subset pairs, that maps 17,000 ICD-9 codes to 141,000 ICD-10 codes and is ambiguous in both directions. Future work is to explore the applicativity of our work beyond ICD, for instance in the context of databases.

Acknowledgements. Thank you to Gadi Aleksandrowicz, Elena Guralnik, Alexander Ivrii, Shiri Moran, Ziv Nevo, Avigail Orni, Julia Rubin, Karen Yorav and anonymous reviewers for important comments on early versions of this work.

References

1. Centers for Disease Control and Prevention, `http://www.cdc.gov/nchs/icd.htm`
2. Centers for Medicare and Medicaid Services, `https://www.cms.gov/Medicare/Coding/ICD10/index.html`
3. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Transactions on Programming Languages and Systems 16(5), 1512–1542 (1994)

4. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press (2001)
5. Eisner, C.: Model Checking the Garbage Collection Mechanism of SMV. Electronic Notes in Theoretical Computer Science 55(3), 289–303 (2001)
6. Eisner, C.: Formal Verification of Software Source Code through Semi-automatic Modeling. Software and System Modeling 4(1), 14–31 (2005)
7. McMillan, K.: Symbolic Model Checking. Kluwer Academic Publishers (1993)
8. RuleBase Parallel Edition, `https://www.research.ibm.com/haifa/projects/verification/RB_Homepage/`

Combining Relational Learning with SMT Solvers Using CEGAR

Arun Chaganty[1], Akash Lal[2], Aditya V. Nori[2], and Sriram K. Rajamani[2]

[1] Stanford
chaganty@stanford.com
[2] Microsoft Research, India
{akashl,adityan,sriram}@microsoft.com

Abstract. In statistical relational learning, one is concerned with inferring the most likely explanation (or *world*) that satisfies a given set of weighted constraints. The weight of a constraint signifies our confidence in the constraint, and the most likely world that explains a set of constraints is simply a satisfying assignment that maximizes the weights of satisfied constraints. The relational learning community has developed specialized solvers (e.g., Alchemy and Tuffy) for such weighted constraints independently of the work on SMT solvers in the verification community. In this paper, we show how to leverage SMT solvers to significantly improve the performance of relational solvers.

Constraints associated with a weight of 1 (or 0) are called *axioms* because they *must* be satisfied (or violated) by the final assignment. Axioms can create difficulties for relational solvers. We isolate the burden of axioms to SMT solvers and only lazily pass information back to the relational solver. This information can either be a subset of the axioms, or even *generalized* axioms (similar to predicate generalization in verification).

We implemented our algorithm in a tool called Soft-Cegar that outperforms state-of-the-art relational solvers Tuffy and Alchemy over four real-world applications. We hope this work opens the door for further collaboration between relational learning and SMT solvers.

1 Introduction

We propose using automated techniques developed in the verification community to improve the efficiency of solving statistical relational learning problems. We first introduce the relational learning problem, review existing solutions, and then present our improvements.

Given data in the form of relations, relational learning involves inferring new relationships that are likely present in the data. For instance, suppose that we are given a set of bibliographic records downloaded from the Internet, and predicates BibAuthor and BibTitle that associate a bibliographic record with its authors and its title, respectively. Because there are variations in how different websites abbreviate author names or paper titles (in addition to spelling mistakes), it may not be immediately clear which records refer to the same paper. A relational learning question is to infer which records (or authors or titles) are the same.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 447–462, 2013.

Such problems naturally involve an interplay between logic and probability. Logic provides the tools to state our intuitions about how new relationships can be derived from existing relationships. For example, we can represent the statement "if two papers have the same authors and the same title, then the two papers are the same" in logic using a formula F as follows:

$$\forall a_0 a_1 b_0 b_1 t_0 t_1.(\mathsf{SameAuthor}(a_0, a_1) \wedge \mathsf{BibAuthor}(a_0, b_0) \wedge \mathsf{BibAuthor}(a_1, b_1) \\ \wedge \mathsf{SameTitle}(t_0, t_1) \wedge \mathsf{BibTitle}(t_0, b_0) \wedge \mathsf{BibTitle}(t_1, b_1)) \Rightarrow \mathsf{SameBib}(b_0, b_1)$$

where SameAuthor, SameTitle and SameBib are the relations that we want to infer. Probability, on the other hand, provides the tools to deal with incompleteness of our models, uncertainty in the world, and errors in the data. Weighted formulae combine both logic and probability. An example of a weighted formula is $0.7 : F$, where the weight 0.7 denotes our confidence in F, i.e., it is our estimate of the probability with which a world satisfies F.

Existing state-of-the-art relational solvers are based on propositional logic. Thus, "structural" constraints such as the fact that SameBib must be an equivalence relation, are encoded using constraints with weight 1 (i.e., they must be satisfied in any world):

$$\begin{array}{ll} \text{Reflexivity: } \forall b_0. & \mathsf{SameBib}(b_0, b_0) \\ \text{Symmetry: } \forall b_0 b_1. & \mathsf{SameBib}(b_0, b_1) \Rightarrow \mathsf{SameBib}(b_1, b_0) \\ \text{Transitivity: } \forall b_0 b_1 b_2. & \mathsf{SameBib}(b_0, b_1) \wedge \mathsf{SameBib}(b_1, b_2) \\ & \Rightarrow \mathsf{SameBib}(b_0, b_2) \end{array} \tag{1}$$

Similarly, the constraint that BibAuthor and BibTitle must encode functions that associate the same author and title with each paper, is specified as follows:

$$\begin{array}{ll} \forall a_0 a_1 b_0 b_1. & (\mathsf{SameBib}(b_0, b_1) \wedge \mathsf{BibAuthor}(a_0, b_0) \wedge \mathsf{BibAuthor}(a_1, b_1)) \\ & \Rightarrow \mathsf{SameAuthor}(a_0, a_1) \\ \forall t_0 t_1 b_0 b_1. & (\mathsf{SameBib}(b_0, b_1) \wedge \mathsf{BibTitle}(t_0, b_0) \wedge \mathsf{BibTitle}(t_1, b_1)) \\ & \Rightarrow \mathsf{SameTitle}(t_0, t_1) \end{array} \tag{2}$$

These constraints affect the scalability of relational solvers in two ways: First, since variables in these formulae are usually universally quantified, and existing relational solvers use propositional logic, these constraints have to be *grounded* [12,16] by instantiating the quantifiers over all constants in the dataset, resulting in many constraints. Second, these are *hard* constraints that must be satisfied, whereas the strength of relational solvers is in dealing with *soft* constraints (because the solvers are optimized to quickly find a good approximation to the ideal result in which violation of some soft constraints is acceptable).

On the other hand, SMT solvers fit the job for precisely solving hard constraints. Moreover, they offer specialized theories that may already capture some of the constraints implicitly. For instance, the fact that SameBib should be an equivalence relation can be captured by defining it as $\mathsf{SameBib}(b_0, b_1) \equiv (f(b_0) = f(b_1))$, where f is some uninterpreted function and "=" is the interpreted equality relation. By leveraging the theory of uninterpreted functions with equality, one can completely elide away the constraints of Formula 1 (and similarly for Formula 2) when using SMT solvers.

Our algorithm proceeds as follows. Let $\mathcal{F}$ be a set of weighted constraints such that $\mathcal{F} = \mathcal{A} \cup \mathcal{F}_s$, where $\mathcal{A}$ is the set of axioms. Let $\mathcal{F}_0 = \mathcal{F}_s$. Inspired by CEGAR (counterexample-guided abstraction refinement), we start by invoking an underlying relational learner (like Tuffy or Alchemy) with the set of formulae $\mathcal{F}_0$. Suppose the relational learner returns a world ω_0. We then check which axioms in $\mathcal{A}$ are violated by ω_0 using SMT solvers, and selectively instantiate axioms on the values of the relations from ω_0 which witness these violations, and add these axioms to $\mathcal{F}_0$ resulting in a larger set of formulae $\mathcal{F}_1$. Next, we invoke the relational learner again with the larger set of formulae $\mathcal{F}_1$, and the iterative process continues until we obtain a world $\hat{\omega}$ that satisfies all the axioms.

While CEGAR is very familiar to the program verification community, and has been used extensively in model checking [1, 2, 6] and in SMT solvers based on DPLL(T) [7, 8], our use of CEGAR for relational learning is new, and requires overcoming several technical challenges.

First, we need to prove that lazily adding axioms does not affect the optimality of the relational learning solution. A key insight here is that satisfied axioms do not contribute to the weight assigned to a world, whereas soft formulae do (we formalize this in Section 2). As a result, if a world ω is optimal for a set of constraints $\mathcal{F}$ and ω happens to satisfy all the axioms in $\mathcal{A}$, then ω is also an optimal world for the set of constraints $\mathcal{F} \cup \mathcal{A}$.

Second, we find that the iterative CEGAR process sometimes requires a large number of iterations, each of which adds formulae forming particular patterns. We propose a technique to detect these patterns and suitably *generalize* the axioms that we add during refinement, thereby greatly reducing the number of iterations needed for convergence.

We have implemented our relational learning algorithm in a tool called Soft-Cegar and evaluated it on well-known applications. We show that Soft-Cegar outperforms state-of-the-art statistical inference tools such as Tuffy [16] and Alchemy [12], both in terms of efficiency and quality of results.

The rest of the paper is organized as follows. Section 2 describes preliminaries and formally defines the problem. Section 3 defines the Soft-Cegar algorithm and proves its correctness. Section 4 describes the empirical evaluation of Soft-Cegar on four applications. Section 5 surveys related work, and Section 6 concludes the paper.

2 Background: Statistical Relational Learning

We are interested in learning relations from a corpus of data given weighted formulae as specifications. The weight of a formula is a real number in the interval $[0, 1]$ that is used to model our confidence in the formula. The corpus of data is called *evidence*, which is an incomplete valuation of the relations. The goal of relational learning is to complete the valuation of the relations and infer a world in order to satisfy the specifications in an optimal manner.

The probabilistic models that we consider for relational learning problems are Markov Logic Networks [20].

2.1 Markov Logic Network

Definition 1. *A Markov Logic Network (MLN) $L = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \rangle$ is a triple, where*

- $\mathcal{D} = \{D_1, D_2, \ldots\}$ *is a set of finite domains.*
- $\mathcal{R} = \{R_1, R_2, \ldots\}$ *is a set of relations over these domains. We assume the existence of a function $\mathcal{S}$ that maps each relation in $\mathcal{R}$ to a schema. For instance, $\mathcal{S}(R_1)$ could be $D_1 \times D_3 \times D_5$, which specifies that R_1 is a three-column relation and $R_1 \subseteq D_1 \times D_3 \times D_5$.*
- $\mathcal{F}$ *is a set of weighted formulae of the form:* $\{w_1 : \forall \bar{x}_1.F_1(\bar{x}_1), w_2 : \forall \bar{x}_2.F_2(\bar{x}_2), \ldots, w_n : \forall \bar{x}_n.F_n(\bar{x}_n)\}$*, where each of the $w_i \in [0, 1]$ are real numbers, and each F_i is a formula in the Domain Relational Calculus (DRC) [22] over universally quantified variables $\bar{x}_i$ and the relations in set $\mathcal{R}$.*

We generalize the schema function $\mathcal{S}$ to both variables and formulas. Given a formula $f = w : \forall \bar{x}.F(\bar{x}) \in \mathcal{F}$, where $\bar{x} = x_1 \cdots x_n$, we define $\mathcal{S}(x_i, f)$ to be the domain of x_i in the formula, and $\mathcal{S}(F, f)$ to be $\mathcal{S}(x_1) \times \cdots \times \mathcal{S}(x_n)$. We will drop the argument f when it is clear from the context.

In the Domain Relational Calculus (DRC), every relation R is viewed as a predicate: $\forall \bar{c} \in \mathcal{S}(R)\ .\ R(\bar{c}) \Leftrightarrow \bar{c} \in R$. We now define the notion of formulae in a DRC. A term is either a constant c or variable x. Atoms are defined as follows:

- If R is a predicate with arity k and $t_1, \ldots, t_k$ are terms, then $R(t_1, \ldots, t_k)$ is an atom.
- If t_1 and t_2 are terms, then $t_1\ \Theta\ t_2$ is an atom, where $\Theta \in \{=, \neq\}$.

Next, we define formulae as follows.

- Every atom is a formula.
- If F_1 and F_2 are formulae, then so are $\neg F_1$, $F_1 \vee F_2$, $F_1 \wedge F_2$ and $F_1 \Rightarrow F_2$.

Figure 1 shows an example MLN $L = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \rangle$ for scheduling classes in a Computer Science department. Section (1) is the `Domain` section which defines the domains $\mathcal{D}$ or attributes of relations in the dataset. These are `Course` (set of courses offered), `Professor` (set of professors in the department), `Slot` (set of slots in a week) and `Student` (the set of students in the department).

Section (2) is the `Relations` section that defines the set of relations $\mathcal{R}$ of interest in the dataset. These are defined as follows.

- $\mathsf{Teaches}(p, c)$: Professor p teaches a course c.
- $\mathsf{Friends}(s_1, s_2)$: Student s_1 is a friend of student s_2.
- $\mathsf{Likes}(s, p)$: Student s likes professor p.
- $\mathsf{NextSlot}(s_1, s_2)$: Slot s_2 immediately follows slot s_1.
- $\mathsf{Attends}(s, c)$: Student s attends course c.
- $\mathsf{Popular}(p)$: Professor p is popular.
- $\mathsf{SameArea}(c_1, c_2)$: Courses c_1 and c_2 are in the same subarea of computer science.
- $\mathsf{HeldIn}(c, s)$: Course c is scheduled in slot s.

(1) Domains:
Course, Professor, Slot, Student
Course = {‘‘Algorithms and Complexity’’, ‘‘Medieval History of Machine Learning’’,...}
Professor = {‘‘Richard Karp’’, ‘‘C.A.R. Hoare’’, ...}
Slot = {‘‘Monday-Wednesday-Friday 9:00-10:00’’, ‘‘Tuesday-Thursday 9:00-10:30’’,... }
Student = { ‘‘Jay Leno’’, ‘‘David Letterman’’, ‘‘Bill Cosby’’, ... }

(2) Relations:
Teaches(Professor, Course), Friends(Student, Student), Likes(Student, Professor), NextSlot(Slot, Slot)
Attends(Student, Course), Popular(Professor), SameArea(Course, Course), HeldIn(Course, Slot)

(3) Weighted formulae:
(* Axiom: Professors and Students cannot be in two places at once *)
1.0: $\forall p_1 c_1 c_2 s_2.\mathsf{Teaches}(p_1, c_1) \wedge \mathsf{Teaches}(p_1, c_2) \wedge \mathsf{HeldIn}(c_1, s_1) \wedge \mathsf{HeldIn}(c_2, s_2) \wedge c_1 \neq c_2 \Rightarrow s_1 \neq s_2$
1.0: $\forall s_1 c_1 c_2 r_1 r_2.\mathsf{Attends}(s_1, c_1) \wedge \mathsf{Attends}(s_1, c_2) \wedge \mathsf{HeldIn}(c_1, r_1) \wedge \mathsf{HeldIn}(c_2, r_2) \wedge c_1 \neq c_2 \Rightarrow r_1 \neq r_2$

(* Axiom: SameArea is an equivalence relation *)
1.0: $\forall c_1 c_2.\mathsf{SameArea}(c_1, c_1)$
1.0: $\forall c_1 c_2.\mathsf{SameArea}(c_1, c_2) \Rightarrow \mathsf{SameArea}(c_2, c_1)$
1.0: $\forall c_1 c_2 c_3.\mathsf{SameArea}(c_1, c_2) \wedge \mathsf{SameArea}(c_2, c_3) \Rightarrow \mathsf{SameArea}(c_1, c_3)$

(* Axiom: Friends is a symmetric relation *)
1.0: $\forall s_1 s_2.\mathsf{Friends}(s_1, s_2) \Rightarrow \mathsf{Friends}(s_2, s_1)$

(* Soft formula: Prefer courses offered by professors you like, and those your friends are taking *)
$0.7 : \forall p_1 c_1 s_1 p_1.\mathsf{Teaches}(p_1, c_1) \wedge \mathsf{Likes}(s_1, p_1) \Rightarrow \mathsf{Attends}(s_1, c_1)$
$0.7 : \forall p_1 c_1 s_1.\mathsf{Teaches}(p_1, c_1) \wedge \mathsf{Popular}(p_1) \Rightarrow \mathsf{Attends}(s_1, c_1)$
$0.7 : \forall s_1 s_2 c_1.\mathsf{Friends}(s_1, s_2) \wedge \mathsf{Attends}(s_1, c_1) \Rightarrow \mathsf{Attends}(s_2, c_1)$

(* Soft formula: Take related courses in same area to gather expertise *)
$0.7 : \forall s_1 c_1 c_2.\mathsf{Attends}(s_1, c_1) \wedge \mathsf{SameArea}(c_1, c_2) \Rightarrow \mathsf{Attends}(s_1, c_2)$

(* Soft formula: Try to schedule classes students attend in consecutive slots
$0.7 : \forall s_1 c_1 c_2 r_1 r_2.\mathsf{Attends}(s_1, c_1) \wedge \mathsf{Attends}(s_1, c_2) \wedge c_1 \neq c_2 \wedge \mathsf{HeldIn}(c_1, r_1) \wedge \mathsf{HeldIn}(c_2, r_2) \Rightarrow \mathsf{NextSlot}(r_1, r_2) \vee \mathsf{NextSlot}(r_2, r_1)$

(* Soft formula: Students tend to take courses in the same area *)
$0.7 : \forall s_1 c_1 c_2.\mathsf{Attends}(s_1, c_1) \wedge \mathsf{Attends}(s_1, c_2) \Rightarrow \mathsf{SameArea}(c_1, c_2)$
$0.7 : \forall s_1 c_1 c_2.\mathsf{Teaches}(s_1, c_1) \wedge \mathsf{Teaches}(s_1, c_2) \Rightarrow \mathsf{SameArea}(c_1, c_2)$

(* Soft formula: Professors are popular if several students like them *)
$0.7 : \forall s_1 p_1.\mathsf{Likes}(s_1, p_1) \Rightarrow \mathsf{Popular}(p_1)$

(4) Evidence:
(* complete values for the relations Teaches, Friends, Likes, and NextSlot*)
Teaches(‘‘Richard Karp’’, ‘‘Algorithms and Complexity Theory’’)
...
Friends(‘‘Jay Leno’’, ‘‘David Letterman’’)
...
Likes(‘‘Jay Leno’’, ‘‘Richard Karp’’)
...
NextSlot(‘‘Monday-Wednesday-Friday 9:00-10:00’’, ‘‘Monday-Wednesday-Friday 10:00-11:00’’)
NextSlot(‘‘Tuesday-Thursday 9:00-10:30’’, ‘‘Tuesday-Thursday 10:30-12:00’’)
...

Fig. 1. An MLN for a Computer Science department together with evidence

Section (3) is the `Weighted formulae` section where all the axioms and soft formulae in $\mathcal{F}$ are defined. Recall that axioms are formulae that must definitely hold. The first two axioms state that professors and students cannot be in two places at the same time. The next set of axioms encode the fact that the relation SameArea is an equivalence relation. In other words, SameArea is a reflexive, symmetric and transitive relation. Finally, we have an axiom that states that the relation Friends is a symmetric relation. The first set of soft formulae specify a student's preference for courses offered by professors that she likes and for the courses taken by her friends. All these formulae are associated with a weight or confidence of 0.7. The next soft formula states that it is likely that students take courses in the same area with the intention of gaining expertise in that area. We also have a soft formula that tries to schedule classes for a student in consecutive slots for the sake of convenience. We have a soft formula that groups two courses into the same area if there are many students who take both courses. Finally, we have a soft formula which says that professors are popular when there are many students who like them.

Section (4) is the `Evidence` section that specifies known relations. The evidence can be thought of as a hard constraint that fixes the values of certain relations. In our dataset, the relations Teaches, Friends, Likes and NextSlot are completely determined. In this example, we are interested in the HeldIn relation, which is a schedule that assigns a course to a slot.

An *axiom* is a weighted formula with weight $w \in \{0, 1\}$. Axioms represent formulae in an MLN that *must* be satisfied or violated. Let $\mathcal{A}(L)$ be the set of axioms in an MLN L.

A *world* of an MLN $L = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \rangle$ is an assignment of all relations in $\mathcal{R}$ according to their schemas. Given a world u and a formula f with weight w, let $\Phi(w, f)$ be w if u satisfies f, and $(1 - w)$ otherwise. The *weight* of a world w, which is an estimate of the likelihood of u is simply defined as: $\prod_{f \in \mathcal{F}} \Phi(u, f)$. The *Maximum a Posteriori Probability* estimate (MAP) is defined as the world with maximum weight. It is possible for an MLN to have multiple MAP solutions; we are only interested in finding one.

Remark. In the verification community, the more common optimization problem is MAXSAT where the objective is to maximize the *sum* of weights of satisfied constraints. If one replaces the weights in an MLN with their logarithms, then solving MAXSAT over such an MLN is similar to computing its MAP. In other words, it is possible to replace relational solvers with MAXSAT solvers. In this paper, however, we only augment relational solvers with SMT solvers. A full comparison of relational solvers to MAXSAT is an interesting direction that we leave as future work.

Under this definition of MAP, we note that negating a weighted formula $w : \forall \bar{x}.F(\bar{x})$ can be done in two ways:

1. Flip the weight, resulting in $f_1 = (1 - w) : \forall \bar{x}.F(\bar{x})$, or
2. Negate the formula, resulting in $f_2 = w : \forall \bar{x}.\neg F(\bar{x})$.

In other words, the formulae f_1 and f_2 are equivalent because using either does not change the MAP. Notice that the universal quantifier does not change to an existential quantifier.

Computing the MAP world of an MLN is NP-hard. However, there are a number of machine learning techniques that efficiently estimate the MAP solution for an MLN given the evidence. For instance, the WalkSAT algorithm [9] that forms the basis of many statistical relational learning tools [12, 16] is one such technique.

The next section shows how we can exploit the distinction between axioms and other soft formulae in MLNs to make exact and approximate MAP estimation more scalable along with a potential for much improved precision.

3 The Soft-Cegar Algorithm

In this section, we describe an algorithm called Soft-Cegar for efficiently computing the MAP for an input MLN L. Soft-Cegar provides a framework for systematically combining any relational MAP inference algorithm with logical inference algorithms that check consistency of the MAP solutions with respect to the axioms defined by $\mathcal{A}(L)$.

We draw inspiration from the following key ideas in program verification and theorem proving: (a) Counterexample-guided abstraction refinement (CEGAR) [6], (b) theorem proving [7], and (c) generalization techniques for accelerating convergence of the CEGAR loop [13].

Notation. For an axiom $f = w : \forall \bar{x}.F(\bar{x})$, let $[\![f]\!]$ be $\forall \bar{x}.F(\bar{x})$ if $w = 1.0$ and $\forall \bar{x}.\neg F(\bar{x})$ if $w = 0.0$. For a set of axioms A, let $[\![A]\!] = \wedge\{[\![f]\!] \mid f \in A\}$. For two sets of axioms A_1 and A_2, we say that A_1 *entails* A_2, or $A_1 \models A_2$ if $([\![A_1]\!] \Rightarrow [\![A_2]\!])$ is valid. Note that if $A_1 = \{1.0 : \forall \bar{x}.F(\bar{x})\}$ and $A_2 = \{1.0 : F(\bar{c})\}$ for some $\bar{c} \in \mathcal{S}(\bar{x})$, then $A_1 \models A_2$.

The Soft-Cegar algorithm is described in Figure 2. The input to the algorithm is an MLN $L = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \rangle$ and its output is a world ω_{MAP} that is a MAP solution of L.

The CEGAR loop of the Soft-Cegar algorithm is described in lines 3–24. The set $\mathcal{F}_{\mathsf{approx}}$ contains all weighted constraints of the MLN, except for its axioms $\mathcal{A}(L)$. We use the set $\mathcal{C}$ to capture a subset of the axioms (or their ground instances). Initially, $\mathcal{C}$ is empty and it grows iteratively inside the CEGAR loop. The algorithm always maintains the invariant that $\mathcal{A}(L) \models \mathcal{C}$.

In line 4, an approximation L_{approx} to the input MLN L is constructed. Note that because $\mathcal{C}$ is entailed by $\mathcal{A}(L)$, L_{approx} always has fewer (or same) number of constraints compared to L. In the first iteration, L_{approx} is the input MLN L without any axioms. Next, in line 6, an off-the-shelf MAP solver $\mathsf{Solve}_{\mathsf{map}}(L_{\mathsf{approx}})$ is invoked on the approximated MLN L_{approx}. This results in a MAP world ω_{approx} of the MLN L_{approx}.

Lines 9–17 check whether ω_{approx} is consistent with the axioms $\mathcal{A}(L)$ of the input MLN L. The set of conflicting axioms, i.e., axioms that are not satisfied by ω_{approx}, are collected in $\mathcal{C}'$. In lines 10–16, for every axiom $w : \forall \bar{x}.F(\bar{x})$, we

```
algorithm SOFT-CEGAR
  input:
    L = ⟨D, R, F⟩: MLN
  output:
    ω_MAP: MAP solution

 1: F_approx := F \ A(L)
 2: C := ∅
 3: loop
 4:    L_approx := ⟨D, R, F_approx ∪ C⟩
 5:    (* Call relational MAP solver *)
 6:    ω_approx := Solve_map(L_approx)
 7:    (* Find conflicting axioms *)
 8:    C' := ∅
 9:    for each w : ∀x̄.F(x̄) ∈ A(L) do
10:       if w = 1.0 then
11:          T := IsConsistent(ω_approx, ¬F)
12:          C' := C' ∪ {1.0 : F(c̄) | c̄ ∈ T}
13:       else
14:          T := IsConsistent(ω_approx, F)
15:          C' := C' ∪ {0.0 : F(c̄) | c̄ ∈ T})}
16:       end if
17:    end for
18:    if C' = ∅ then
19:       ω_MAP := ω_approx
20:       return ω_MAP
21:    end if
22:    C := C ∪ C'
23:    C := Generalize(C, L)
24: end loop
```

Fig. 2. The SOFT-CEGAR algorithm

```
algorithm GENERALIZE
  input:
    C: a set of axioms L : MLN
  parameters:
    thresh: a floating point number
  output:
    G: a set of axioms such that A(L) ⊨ G ⊨ C

 1: G := ∅
 2: for all φ ∈ A(L) do
 3:    let φ = 1.0 : ∀x̄F(x̄)
 4:    for all partial instantiations ψ of φ do
 5:       let covered := {f ∈ C | {ψ} ⊨ {f}}
 6:       let support := |covered|/|S(F)|
 7:       if support > thresh then
 8:          (* Generalize *)
 9:          G := G ∪ {ψ}
10:          C := C\covered
11:          break
12:       end if
13:    end for
14: end for
15: return G ∪ C
```

Fig. 3. The GENERALIZE algorithm

compute a set of tuples $\mathcal{T} \subseteq \mathcal{S}(F)$ in ω_{approx} that violate the axiom (recall that $\mathcal{S}(F)$ is the product of the domains of the variables in F, as defined in Section 2). This is computed by the procedure IsConsistent called on lines 11 and 14. This is essentially a call to an SMT solver. The set of conflict axioms $\mathcal{C}'$ is the set of all instances of the axiom $w : \forall \bar{x}.F(\bar{x})$ over the set of tuples $\mathcal{T}$. This is computed in lines 11 and 14.

If the set $\mathcal{C}'$ is empty, this means that the current solution ω_{approx} respects all axioms in $\mathcal{A}(L)$ and the procedure stops and returns $\omega_{\mathsf{MAP}} = \omega_{\mathsf{approx}}$ (line 20). Otherwise, the set of weighted formulae is refined with $\mathcal{C}'$ (line 22). It is easy to see that $\mathcal{A}(L) \models \mathcal{C}'$ because the latter only contains ground instances of axioms. Thus, when $\mathcal{C}$ is updated on line 22, we still have $\mathcal{A}(L) \models \mathcal{C}$.

In order to accelerate the convergence of the CEGAR loop, we make use of a generalization procedure (line 23). One can use any procedure as long as it satisfies the following condition: given argument $\mathcal{C}$, it must return a set A such that $\mathcal{A}(L) \models A \models \mathcal{C}$. Note that returning $\mathcal{C}$ itself is always a valid solution, but it does not accelerate convergence. Similarly, returning $\mathcal{A}(L)$ is also a valid solution, but this is too big a jump that may place a huge burden on the underlying solver. The particular implementation of generalization that we use chooses a middle ground. It is described in Figure 3 and the details will follow in Section 3.1.

Example. Consider the SameArea equivalence relation in Figure 1. Without the axiom of transitivity present in $\mathcal{F}_{\mathsf{approx}}$, a possible world of L_{approx} (on line 7) may have both the tuples SameArea("Static Analysis", "Program Analysis") and SameArea("Program Analysis", "Abstract Interpretation"), but not the tuple SameArea("Static Analysis", "Abstract Interpretation"). In this case, the set of tuples $\mathcal{T}$ (on line 10) returned by the SMT solver contains the tuple $\bar{c}$ = ("Static Analysis", "Program Analysis", "Abstract Interpretation"), and the conflict $F(\bar{c})$ added on line 11 is:
SameArea("Static Analysis", "Program Analysis") $\wedge$
SameArea("Program Analysis", "Abstract Interpretation")
$\Rightarrow$ SameArea("Program Analysis", "Abstract Interpretation")

This axiom prevents the MAP solver from choosing such a world in future iterations.

A property of the Soft-Cegar algorithm is that the laziness does not cause us to sacrifice precision. This is because satisfied axioms do not contribute to the weight assigned to a world. Thus, as long as all the axioms are satisfied, the weight of a world in an MLN with or without axioms is the same. We still need to establish that there is no other world that can satisfy all the axioms and have a higher weight. Assuming that the underlying relational learner is optimal, we can argue that such a situation cannot arise. Below, we exploit this natural separation that exists between soft formulas and axioms to show that the Soft-Cegar algorithm is able to compute an exact MAP solution, i.e., a world with the maximum weight.

Theorem 1. *The* Soft-Cegar *algorithm returns an exact MAP solution provided the MAP solver* $\mathsf{Solve}_{\mathsf{map}}$ *always returns exact MAP solutions.*

Proof: For simplicity, assume that axioms always have the weight 1.0 (otherwise, replace the axiom $0.0 : \forall \bar{x}.F(\bar{x})$ with $1.0 : \forall \bar{x}.\neg F(\bar{x})$). Recall the definition of the *weight* of a world from Section 2. Define the *weight* of an MLN to be the weight of its MAP solution, i.e., the maximum possible weight of a world in the MLN. Consider two MLNs $L_1 = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \cup \mathcal{C}_1 \rangle$ and $L_2 = \langle \mathcal{D}, \mathcal{R}, \mathcal{F} \cup \mathcal{C}_2 \rangle$ such that all the formulae in $\mathcal{C}_1$ and $\mathcal{C}_2$ are axioms with weight 1.0 and $\mathcal{C}_2 \models \mathcal{C}_1$. Then $\mathit{weight}(L_1) \geq \mathit{weight}(L_2)$ because:

A. Let ω be a world with weight $w \neq 0$ in L_2. Then ω must satisfy $\mathcal{C}_2$. Because $\mathcal{C}_2$ entails $\mathcal{C}_1$, ω satisfies $\mathcal{C}_1$. Because L_1 and L_2 have the same set of weighted constraints (barring axioms), the weight of ω in L_1 must also be w.
B. Let u be a MAP of L_2, i.e., its weight in L_2 is $\mathit{weight}(L_2)$. From A, the weight of u in L_1 is also $\mathit{weight}(L_2)$. (Nothing to prove if $\mathit{weight}(L_2) = 0$.)
C. If there is a world with weight w in L_1 then, by definition, $\mathit{weight}(L_1) \geq w$. Thus, from B, $\mathit{weight}(L_1) \geq \mathit{weight}(L_2)$.

Next, if u' is a MAP of L_1 and u' satisfies $\mathcal{C}_2$, then the weight of u' in L_2 is $\mathit{weight}(L_1)$. This implies $\mathit{weight}(L_1) = \mathit{weight}(L_2)$ and u' is a MAP of L_2.

The proof of the theorem follows from these observations. Let L_1 be the MLN L_{approx} on the last iteration of the Soft-Cegar algorithm. Then L_1 is

$\langle \mathcal{D}, \mathcal{R}, \mathcal{F}_{\mathsf{approx}} \cup \mathcal{C} \rangle$ for some $\mathcal{C}$ that is accumulated in the various iterations of the loop. And the input MLN L can be written as $L_2 = \langle \mathcal{D}, \mathcal{R}, \mathcal{F}_{\mathsf{approx}} \cup \mathcal{A}(L) \rangle$. Because $\mathcal{A}(L) \models \mathcal{C}$, when $\mathcal{C}' = \emptyset$ on line 20 we know that ω_{approx} satisfies all the axioms $\mathcal{A}(L)$. In this case, the MAP solution of L_1 must be an MAP solution of L_2. ■

Theorem 1 assumes that the underlying MAP solver is exact. In practice, existing MAP solvers are based on probabilistic and approximation algorithms and cannot handle axioms precisely. As we show in the next section, even with these imprecise MAP solvers, SOFT-CEGAR's ability to handle axioms specially allows it to improve both runtime and precision of the inference.

3.1 Generalization

In this section, we describe the generalization procedure GENERALIZE employed by SOFT-CEGAR. As in program verification, the goal of generalization is to reduce the number of CEGAR iterations needed, while not imposing a huge burden on the underlying solver. We first work through an example.

Consider the course scheduling MLN from Figure 1. We assume that the following facts are part of the world ω_{approx} in some arbitrary iteration i of SOFT-CEGAR.

$\mathsf{Attends}(\text{Student1}, \text{Course1})$	$\mathsf{HeldIn}(\text{Course1}, \text{Slot1})$
$\mathsf{Attends}(\text{Student1}, \text{Course3})$	$\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$

This world violates the following axiom that states that students cannot be in two places at the same time.

$$1.0 : \forall s_1 c_1 c_2 r_1 r_2 . \mathsf{Attends}(s_1, c_1) \wedge \mathsf{Attends}(s_1, c_2) \wedge \mathsf{HeldIn}(c_1, r_1) \wedge \mathsf{HeldIn}(c_2, r_2) \wedge c_1 \neq c_2 \Rightarrow r_1 \neq r_2$$

In order, to rule out this world ω_{approx}, the following conflict axiom is added to the set of weighted formulae for the approximate MLN in next iteration of SOFT-CEGAR.

$$1.0 : \neg\mathsf{Attends}(\text{Student1}, \text{Course1}) \vee \neg\mathsf{Attends}(\text{Student1}, \text{Course3}) \vee \neg\mathsf{HeldIn}(\text{Course1}, \text{Slot1}) \vee \neg\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$$

However, resolving conflicts at such a fine granularity might be lead to a prohibitively large number of iterations and as a result, the following facts that violate the same axiom above might show up in worlds computed by subsequent iterations of SOFT-CEGAR.

i	$\mathsf{Attends}(\text{Student1}, \text{Course1})$	$\mathsf{Attends}(\text{Student1}, \text{Course3})$
	$\mathsf{HeldIn}(\text{Course1}, \text{Slot1})$	$\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$
$i+1$	$\mathsf{Attends}(\text{Student2}, \text{Course1})$	$\mathsf{Attends}(\text{Student2}, \text{Course3})$
	$\mathsf{HeldIn}(\text{Course1}, \text{Slot1})$	$\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$
$i+2$	$\mathsf{Attends}(\text{Student3}, \text{Course1})$	$\mathsf{Attends}(\text{Student3}, \text{Course3})$
	$\mathsf{HeldIn}(\text{Course1}, \text{Slot1})$	$\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$
$i+3$	$\mathsf{Attends}(\text{Student4}, \text{Course1})$	$\mathsf{Attends}(\text{Student4}, \text{Course3})$
	$\mathsf{HeldIn}(\text{Course1}, \text{Slot1})$	$\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$

Furthermore, it is possible that this sequence of conflict axioms could continue for many iterations. There could be several reasons for this behavior such as

the popularity of Course1 and Course3. Therefore, we would like to find a more "general" conflict axiom that entails many possible conflict axioms in future iterations so as to reduce the overall number of iterations of Soft-Cegar. For instance, each of the following two axioms rule out all of the violating facts mentioned before.

$1.0 : \forall x.\neg\mathsf{Attends}(x, \text{Course1}) \vee \neg\mathsf{Attends}(x, \text{Course3}) \vee$ $\neg\mathsf{HeldIn}(\text{Course1}, \text{Slot1}) \vee \neg\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$
$1.0 : \forall xy.\neg\mathsf{Attends}(x, y) \vee \neg\mathsf{Attends}(x, \text{Course3}) \vee$ $\neg\mathsf{HeldIn}(y, \text{Slot1}) \vee \neg\mathsf{HeldIn}(\text{Course3}, \text{Slot1})$

In general, there can be many choices of generalized axioms. We have to balance the amount of generalization with the number of iterations.

Our generalization algorithm searches over the space of *partial instantiation* of axioms. A partial instantiation of an axiom is one in which some (or all) of the quantified variables of the axiom are ground to particular constants. For instance, $1.0 : \forall xz.F(x, c, z)$ is a partial instantiation of $1.0 : \forall xyz.F(x, y, z)$, where $c \in \mathcal{S}(y)$. An abstract description of our algorithm is shown in Figure 3. It is controlled by a threshold value *thresh* that lies between 0 and 1. The algorithm works by searching over the space of all partial instantiations of axioms, looking for one with a support higher than *thresh*. Here, support of an axiom f is defined as the fraction of formulae in $\mathcal{C}$ that entail f, divided by its total number of instantiations. (The division prevents over generalization.) If one such instantiation is found, we add it to $\mathcal{G}$ and remove the covered axioms from $\mathcal{C}$.

Soft-Cegar uses an efficient implementation of this algorithm. It uses a *thresh* value obtained by training the algorithm on small datasets.

4 Evaluation

In this section, we describe our implementation of Soft-Cegar and compare it with two state-of-the-art relational learning engines Alchemy [12] and Tuffy [16] on four real world applications. All experiments were performed on a 2.66 GHz Intel Xeon quad core processor system with 16 GB RAM running Microsoft Windows Server 2008. Soft-Cegar is implemented in F# and uses the PostgreSQL 8.4 RDBMS engine for storing and manipulating relations. The implementation of Soft-Cegar uses Tuffy as the underlying relational MAP solver (implementation of $\mathsf{Solve}_{\mathsf{map}}$ subroutine in Figure 2). We consider four real-world applications for our evaluation.

Advisor Recommendation (AR). The MLN for this application specifies a recommendation system for starting graduate students that helps them find good PhD advisors. This recommendation is made based on the interests of students (expressed using the Likes(Person, Paper) relation) and weighted formulae such as "prefer an advisor who has graduated a student in an area of interest". The axioms consists of rules such as "the advisor should have had at least one student who is the first author of a paper", in addition to axioms that

specify the theory of equality used to discover equivalence classes over papers. The dataset (AIDB) for this application was created from the AI genealogy project (`http://aigp.eecs.umich.edu`) for advisor-student relationships and DBLP (`http://dblp.uni-trier.de`) for bibliographic information. This dataset consists of information about 25 researchers and 600 papers.

Entity Resolution (ER) [23]: This is the problem of identifying duplicate entities in a database. We use the Cora dataset [3] (available at `http://alchemy.cs.washington.edu/data/cora`) that consists of 1295 citations and 132 distinct research papers for our experiments. The objective is to find identical authors, venues, titles and citations in the database. The MLN for this task is described in (`http://alchemy.cs.washington.edu/mlns/er`). Since this is essentially a classification problem, most of the axioms in the MLN are those specifying an equivalence relation.

Information Extraction (IE) [18]: The task for this application is to extract database records from text or semi-structured sources. In our experiment, we use the Cora dataset and the MLN (available at `http://alchemy.cs.washington.edu/mlns/ie`) specifies extraction of author, title and venue fields from this bibliographic dataset. The axioms in this MLN specify the theory of uninterpreted functions [4].

Relational Classification (RC) [16]: In this application, papers in the dataset are classified based on categories of papers published by same authors. The axioms in this MLN specify the theory of equality and uninterpreted functions [4].

The statistics for these four applications (MLNs) and their corresponding datasets is shown in Table 1.

Experiments. Table 2 summarizes the results obtained by running Soft-Cegar, Tuffy and Alchemy over the four applications. The runtime is in

Table 1. Application MLN and dataset statistics

	AR	ER	IE	RC
#relations	14	14	19	5
#formula	24	3.8K	1.1K	32
#axioms	6	7	3	2
#atoms	88K	20K	81K	9860
#evidence-atoms	65K	676	613K	430K
#query-atoms	188	400	400	400

Table 2. Empirical evaluation of Soft-Cegar

Method	Iterations	Time	Solution Cost
Advisor Recommendation			
Soft-Cegar	18	06:44	3669.50
Tuffy	1	-	*
Alchemy	-	-	*
Entity Resolution			
Soft-Cegar	8	13:06	28112.24
Tuffy	1	15:13	34416.97
Alchemy	1	16:17	5287838.62
Information Extraction			
Soft-Cegar	3	17:46	109.40
Tuffy	1	55:49	3944.29
Alchemy	-	-	*
Relational Classification			
Soft-Cegar	2	05:00	870.37
Tuffy	1	05:42	874.63
Alchemy	-	-	*

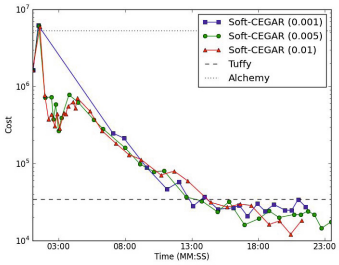


Fig. 4. Cost of solution vs. time for ER on the Cora dataset

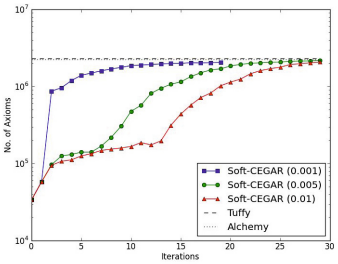


Fig. 5. Number of axioms vs. number of iterations for ER on the Cora dataset

minutes:seconds and the cost of a world ω is proportional to negative log of its weight (minimizing this quantity is equivalent to finding the MAP solution).

The rows for Soft-Cegar, Tuffy and Alchemy in Table 2 report the runtimes and solution costs obtained by running these tools over the four applications. The "*" entries in the table correspond to an out-of-memory exception result. For instance, Tuffy runs out of memory on the AR application (after 4 hours), while Alchemy runs out of memory on the IE and RC applications (after 18 hours). On the other hand, Soft-Cegar completes and quickly produces a result on all applications. On the applications where Tuffy and Alchemy terminate and produce a result, it is important to also note that Soft-Cegar produces solutions which are superior (lower cost) in terms of quality. This empirically supports our intuition that reducing the axiom burden on a relational solver via CEGAR can lead to improvements in efficiency as well as precision.

Figure 4 plots solution cost vs. time for Soft-Cegar, Tuffy and Alchemy on the ER application over the Cora dataset. We have three curves for Soft-Cegar, one for each $thresh = 0.001$, $thresh = 0.005$ and $thresh = 0.01$, where $thresh$ is a parameter that controls the degree of generalization. A lower value of $thresh$ corresponds to more generalization and therefore better acceleration of the CEGAR loop. It can be seen for all curves that the solution cost decreases with time and that Soft-Cegar performs much better than both Tuffy and Alchemy. It can also be seen from Figure 5 that the rate of addition of axioms decreases with increase in values of $thresh$. Note that that Soft-Cegar results in lower cost solutions, validating our intuition that reducing the burden on the relational MAP solver via lazy addition of axioms can also improve the quality of the results. As expected, Tuffy performs much better than Alchemy for this application and this is consistent with the results reported in [16].

For the IE application, Soft-Cegar completes in 3 iterations and this can be attributed to effective generalization which also results in a more precise solution than Tuffy. For the RC application, Soft-Cegar completes in two iterations and this is due to a small number of axioms in this application.

5 Related Work

Several optimizations for scaling statistical relational learning tools have been proposed recently. *Lifted inference* [5, 15, 17] exploits the structure of graphical models to efficiently perform message passing and compute marginal probabilities. In particular, subsets of components that will send and receive identical messages during belief propagation are identified and grouped together for efficiency. *Lazy inference* [19] exploits the observation that most variables in inferred worlds typically have default values (a value that is much more frequent than the others). Thus, we can save both time and memory by assuming that almost all of variables have default values, and gradually refine the variables whose values need to be changed to get an optimum score. Lazy inference generalizes earlier work on Lazy-WalkSAT [24]. *Lazy grounding* approaches have been devised to deal with the size of domains, which are typically very large, by lazily adding constants to be considered from each domain, on demand. Coarse-to-fine inference [10] groups constants from domains into *types* and refines the types progressively to get better scores for the inferred solution. Cutting plane inference [21] starts with a *partial grounding* (which considers only a subset of constants from each domain) with a *partial score*, and iteratively ascertains which constants to add to the grounding so as to improve the score of the inferred solution. The Tuffy [16] tool uses relational database techniques to efficiently perform grounding and avoid constructing groundings that do not matter for calculating the score of the final inferred world.

Our technique for handling axioms lazily is complementary and orthogonal to all these existing optimizations. Tools for relational learning such as Alchemy [12] and Tuffy [16] are publicly available with benchmark suites. Our work was inspired by trying out these tools on these benchmark suites, and observing that CEGAR techniques can significantly improve the performance of these tools. We have implemented our CEGAR based relational learning algorithm in a tool called Soft-Cegar using Tuffy as the underlying relational learner. Tuffy is the most recent optimized relational learning solver, and it incorporates all the optimizations mentioned above from the relational learning community. Our experimental results show that Soft-Cegar outperforms Tuffy, giving empirical evidence that lazily instantiating axioms with CEGAR gives substantial gains over and above all the above optimizations. In addition, our work enables enriching the language of relational learning tools with SMT theory reasoning. Though all our current examples use only the EUF theory, our algorithm works with other theories such as linear arithmetic, which are not currently available in relational learners.

6 Conclusion

Inspired by the wide use of CEGAR (Counterexample Guided Abstraction Refinement) in program verification, we proposed a technique to lazily add axioms in the context of statistical relational inference. We proved that the technique

is guaranteed to yield optimal answers assuming that the underlying relational learner is optimal (Theorem 1). In practice, even though existing statistical relational learners use heuristics, and are far from being optimal, we empirically validated that our lazy addition of axioms greatly improves both their runtime and the quality of their solution. The current implementation of our algorithm uses an existing relational learner as a blackbox, and is able to make use of all the orthogonal and complementary optimizations that have been already developed and implemented in existing relational learning tools.

We see several avenues for future work. Our CEGAR framework can be used to add expressiveness to the formulae used in relational learning. In particular, it is possible to integrate theories supported by SMT solvers such as linear arithmetic and provide a more expressive language of formulas for MLNs. This would enable inference to infer constants in the solution that are not present in the evidence or in the domains. While the optimality provided by Theorem 1 is easy to prove for finite domains with such an extension, more work is needed to guarantee optimality and handle infinite domains. Another direction worth exploring is the problem of learning axioms from data using logical reasoning algorithms, inspired by recent approaches proposed to infer the structure of graphical models from data [11, 14].

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: Principles of Programming Languages (POPL 2002), pp. 1–3 (2002)
2. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The Software Model Checker Blast. STTT: International Journal on Software Tools for Technology Transfer 9(5-6), 505–525 (2007)
3. Bilenko, M., Mooney, R.: Adaptive duplicate detection using learnable string similarity measures. In: Knowledge Discovery and Data Mining (KDD 2003), pp. 39–48 (2003)
4. Bradley, A.R., Manna, Z.: The Calculus of Computation: Decision Procedures with Applications to Verification. Springer (2007)
5. Braz, R., Amir, E., Roth, D.: Lifted first-order probabilistic inference. In: International Joint Conference on Artificial Intelligence (IJCAI 2005), pp. 1319–1325 (2005)
6. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 154–169. Springer, Heidelberg (2000)
7. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Flanagan, C., Joshi, R., Ou, X., Saxe, J.B.: Theorem proving using lazy proof explication. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 355–367. Springer, Heidelberg (2003)
9. Kautz, H., Selman, B., Jiang, Y.: A general stochastic approach to solving problems with hard and soft constraints. In: Gu, D., Du, J., Pardalos, P. (eds.) The Satisfiability Problem: Theory and Applications, pp. 573–586. AMS (1997)

10. Kiddon, C., Domingos, P.: Coarse-to-fine inference and learning for first-order probabilistic models. In: National Conference on Artificial Intelligence, AAAI 2011 (2011)
11. Kok, S., Domingos, P.: Learning the structure of Markov logic networks. In: International Conference on Machine Learning (ICML 2005), pp. 441–448 (2005)
12. Kok, S., Sumner, M., Richardson, M., Singla, P., Poon, H., Lowd, D., Domingos, P.: The Alchemy system for statistical relational AI. Technical report, University of Washington (2007), `http://alchemy.cs.washington.edu`
13. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
14. Mihalkova, L., Mooney, R.: Bottom-up learning of Markov logic network structure. In: International Conference on Machine Learning (ICML 2007), pp. 625–632 (2007)
15. Milch, B., Zettlemoyer, L.S., Kersting, K., Haimes, M., Kaelbling, L.P.: Lifted probabilistic inference with counting formulas. In: National Conference on Artificial Intelligence (AAAI 2008), pp. 1062–1068 (2008)
16. Niu, F., Re, C., Doan, A., Shavlik, J.: Tuffy: Scaling up statistical inference in Markov logic networks using an RDBMS. In: International Conference on Very Large Data Bases, VLDB 2011 (2011)
17. Poole, D.: First-order probabilistic inference. In: International Joint Conference on Artificial Intelligence (IJCAI 2003), pp. 985–991 (2003)
18. Poon, H., Domingos, P.: Joint inference in information extraction. In: National Conference on Artificial Intelligence (AAAI 2007), pp. 913–918 (2007)
19. Poon, H., Domingos, P., Sumner, M.: A general method for reducing the complexity of relational inference and its application to mcmc. In: National Conference on Artificial Intelligence, AAAI 2008 (2008)
20. Richardson, M., Domingos, P.: Markov logic networks. Machine Learning 62, 107–136 (2006)
21. Riedel, S.: Cutting plane map inference for Markov logic. In: Statistical Relational Learning, SRL 2009 (2009)
22. Silberschatz, A., Korth, H., Sudarshan, S.: Database Systems Concepts, 5th edn. McGraw-Hill, Inc. (2006)
23. Singla, P., Domingos, P.: Entity resolution with Markov logic. In: International Conference on Data Mining (ICDM 2006), pp. 572–582 (2006)
24. Singla, P., Domingos, P.: Memory-efficient inference in relational domains. In: National Conference on Artificial Intelligence (AAAI 2006), pp. 488–493 (2006)

A Fully Verified Executable LTL Model Checker*

Javier Esparza[1], Peter Lammich[1], René Neumann[1], Tobias Nipkow[1], Alexander Schimpf[2], and Jan-Georg Smaus[3]

[1] Technische Universität München
{esparza,lammich,neumannr,nipkow}@in.tum.de
[2] Universität Freiburg
schimpfa@informatik.uni-freiburg.de
[3] IRIT, Université de Toulouse
smaus@irit.fr

Abstract. We present an LTL model checker whose code has been completely verified using the Isabelle theorem prover. The checker consists of over 4000 lines of ML code. The code is produced using recent Isabelle technology called the Refinement Framework, which allows us to split its correctness proof into (1) the proof of an abstract version of the checker, consisting of a few hundred lines of "formalized pseudocode", and (2) a verified refinement step in which mathematical sets and other abstract structures are replaced by implementations of efficient structures like red-black trees and functional arrays. This leads to a checker that, while still slower than unverified checkers, can already be used as a trusted reference implementation against which advanced implementations can be tested. We report on the structure of the checker, the development process, and some experiments on standard benchmarks.

1 Introduction

Developers of verification tools are often asked if they have verified their own tool. The question is justified: verification tools are trust-multipliers—they increase our confidence in the correctness of many other systems—and so their bugs may have a particularly dangerous multiplicative effect. However, with the current state of verification technology, proving software correct is dramatically slower than testing it. The strong advances in verification technology of the last two decades would have not been possible if verifiers had only deployed verified tools.

In this paper we propose a pragmatic solution to this dilemma, precisely because of the advances in verification: verified *reference implementations* of standard verification services. Verifiers working on sophisticated techniques to increase efficiency can test their tools against the reference implementation, and so gain confidence in the correctness of their systems. We present a reference implementation for an LTL model checker for finite-state systems à la SPIN [10]. The model checker follows the well-known automata-theoretic approach. Given a finite-state program P and a formula ϕ, two Büchi automata are constructed that recognize the executions of P, and all potential executions of P that violate

* Research supported by DFG grant CAVA, *Computer Aided Verification of Automata.*

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 463–478, 2013.

ϕ, respectively. The latter is constructed using the algorithm of Gerth et al. [7], which—while not the most efficient—is particularly suitable for formal proof because of its inductive structure. Then the product of the two automata is computed and tested on-the-fly for emptiness. For the emptiness check we use an improved version of the nested depth-first search algorithm [6,22].

A reference implementation of an LTL model checker must be *fully* verified (i. e. it must be proved that the program satisfies a *complete* functional specification). At the same time, it must be reasonably efficient—as a rule of thumb, the verifier should be able to run a test on a medium-size benchmark in seconds or at most a few minutes. Simultaneously meeting these two requirements poses big challenges. An imperative programming style allows one to use efficient update-in-place and random-access data structures, like hash tables; however, producing fully verified imperative code is hard. In contrast, a functional style, due to its extensive use of recursion and recursively defined data structures like lists and trees, allows for standard proofs by induction; however, efficiency can be seriously compromised. In this paper we choose a functional style (we produce ML code), and overcome the efficiency problem by means of a development process based on refinement. We use the Isabelle Collection and Refinement Frameworks [13,16] presented in Section 5. The Refinement Framework allows us to first prove correct an inefficient but simple formalization and then refine it to an efficient version in a stepwise manner. The Collection Framework provides a pre-proved library of efficient implementations of abstract types like sets by red-black trees or functional versions of arrays, which we use as replacement of hash tables.

To prove full functional correctness of executable code, we define the programs in the logic HOL of the interactive theorem prover Isabelle [19]. More precisely, the programs are defined in a subset of HOL that corresponds to a functional programming language. After proving the programs correct, ML (or OCaml, Haskell or Scala) code can be generated *automatically* from those definitions [8]; like in PVS and Coq, the code generator (which translates equations in HOL into equations in some functional programming language) is part of the trusted kernel of Isabelle. Isabelle proofs are trustworthy because Isabelle follows the LCF architecture and all proofs must go through a small kernel of inference rules.

We conduct some experiments on standard benchmarks to check the efficiency of our code. To gain a first impression, for products with 10^6–10^7 states, our implementation explores 10^4–10^5 states per second. While this is still below the efficiency of the fastest LTL model checkers, it fulfills the goal stated above: products with 10^6–10^7 states allow to explore many corner cases of the test space.

All supporting material, including the ML code, can be found online at `http://cava.in.tum.de/CAV13`.

1.1 Related Work

To the best of our knowledge, we present the first fully verified executable LTL model checker. In previous work [21], we had presented a formalization of the LTL-to-Büchi translation by Gerth et al. [7] in Isabelle. However, instead of sets,

lists were used everywhere for executability reasons. In the present work, we have re-formalized it using the Refinement Framework, in order to separate the abstract specification (in terms of sets) and the concrete implementation (refining sets to red-black trees, ordered lists etc., as appropriate). This increased modularity and performance. Moreover we have added a second phase that translates the resulting generalized Büchi automata into ordinary ones.

The first verification of a model checker we are aware of is by Sprenger for the modal μ-calculus in Coq [23]. No performance figures or larger examples are reported.

There is a growing body of verified basic software like a C compiler [17] or the seL4 operating system kernel [11]. With Leroy's compiler we share the functional programming approach. In contrast, seL4 is written in C for performance reasons. But verifying its 10,000 lines required 10–20 person years (depending on what you count). We believe that verifying a model checker at the C level would require many times the effort of our verification, and that verifying functional correctness of significant parts of SPIN is not a practical proposition today. Even seL4, although very performant, was designed with verification in mind.

Of course the idea of development by refinement is an old one (see Section 5 for references). A popular incarnation is the B-Method [1] for which a number of support tools exist. The main difference is that B aims at imperative programs while we aim at functional ones.

2 Isabelle/HOL

Isabelle/HOL [19] is an interactive theorem prover based on Higher-Order Logic (HOL). You can think of HOL as a combination of a functional programming language with logic. Although Isabelle/HOL largely follows ordinary mathematical notation, there are some operators and conventions that should be explained. Like in functional programming, functions are mostly curried, i. e. of type $\tau_1 \to \tau_2 \to \tau$ instead of $\tau_1 \times \tau_2 \to \tau$. This means that function application is usually written $f\ a\ b$ instead of $f(a, b)$. Lambda terms are written in the standard syntax $\lambda x.\ t$ (the function that maps x to t) but can also have multiple arguments $\lambda x\ y.\ t$, paired arguments $\lambda(x,y).\ t$, or dummy arguments $\lambda\text{-}.\ t$. Names may contain hyphens, as in *nested-dfs*; do not confuse them with subtraction.

Type variables are written *'a*, *'b* etc. Compound types are written in postfix syntax: τ *set* is the type of sets of elements of type τ, similarly for τ *list*. Lists come with the standard functions *length*, *set* (converts a list into a set), *distinct* (tests if all elements are distinct), and *xs!i* (returns the *i*th element of list *xs*). The function *insert* inserts an element into a set.

A record with fields $l_1, \ldots, l_n$ that have the values $v_1, \ldots, v_n$ is written $(\!|l_1 = v_1, \ldots, l_n = v_n|\!)$. The field l of a given record r is selected just by function application: $l\ r$.

In some places in the paper we have simplified formulas or code marginally to avoid distraction by syntactic or technical details, but in general we have stayed faithful to the sources.

3 A First View of the Model Checker

In this section we give an overview of the checker and its correctness proof for non-specialists in interactive theorem proving. The checker consists of about 4900 lines of ML. The input consists of a system model and an LTL formula. The modeling language, which we call *Boolean programs*, is a simple guarded command language with Booleans as datatype. The atomic propositions of the formula are of the form `x`, stating that variable `x` is currently true.

Boolean programs are compiled using a function that translates them into an interpreted assembly program, also defined within Isabelle, with a simple notion of configuration. We then call *cava-code*, the main function of the model checker; applied to a compiled Boolean program *bpc* (more precisely, *bpc* is always a pair consisting of a compiled program and an initial configuration) and a formula *phi*, *cava-code* returns either *NO-LASSO* or *LASSO y to-y cyc*. *NO-LASSO* means that the product automaton contains no accepting lasso, i. e. that every execution of the program satisfies the property. *LASSO y to-y cyc* describes an accepting lasso, which corresponds to a counter-example, i. e. an execution of the program violating *phi*: *y* is an accepting state, *to-y* is a path leading to it (given as a list of states), and *cyc* is a cycle from *y* to *y*. The generated ML code looks as follows:

```
fun cava-code bpc phi =
  let val y = LTL-to-BA-code (LTLcNeg phi);
      val x = (graph-Delta-code bpc y, graph-F-code y);
  in nested-dfs-code x start-node end;
```

The main subfunctions already appear in the text of *cava-code*:

- *LTL-to-BA-code phi* is based on the tableau construction by Gerth et al. [7]. The function takes an LTL formula as input and returns the initial state, transition function, and acceptance condition of a Büchi automaton for the formula *phi*. The construction of [7] proceeds by recursion on the structure of the formula, which makes it particularly suitable for verification.
- *graph-Delta-code bpc y* returns the transition function of the product of a Büchi automaton recognizing the runs of *bpc* and the Büchi automaton *y*. *graph-F-code y* returns the accepting states of the product. (As the Büchi automaton of *bpc* has only final states, it is not required as a parameter of *graph-F-code*.)
- *nested-dfs-code x start-node* implements the algorithm of [22] for emptiness of Büchi automata. This algorithm is an improvement of the nested depth-first search algorithm of [9], which in turn improves on the original nested depth-first search algorithm of [6].

Function *LTLcNeg phi* returns the negation of *phi*. So the program negates the formula, computes a Büchi automaton for it, intersects it with a Büchi automaton for the Boolean program, and checks for emptiness.

The model checker and its correctness proof are developed in three steps, using Isabelle's Refinement Framework [15,16], which is described in Section 5. Here, we give a brief overview. Each function *foo-code* in the final ML code is the result

of a three-step process. We first formalize an abstract function *foo*, together with its specification. Abstract functions are allowed to use nondeterministic choice, abstract sets as data structures, etc.; they can be seen as formalized pseudocode. The abstract functions comprise about 250 lines of Isabelle code, which can be found in `http://cava.in.tum.de/files/CAV13/abstract-functions.pdf`. The first proof step consists of showing that they satisfy their specifications.

In a second step, we formalize a *foo-code* function based on *foo*. Here, operations on sets are replaced by corresponding operations on red-black trees or arrays. For instance, an instruction like "**let** $X' = X \cup \{x\}$" is replaced by an insert operation on, say, red-black trees. The second proof step consists of proving that *foo-code* is a refinement of *foo*. Loosely speaking, this means that the result of the (deterministic) *foo-code* is one of the results of the (usually nondeterministic) *foo*. The second step does not significantly increase the code length.

Finally, *foo-code* is automatically transformed into ML code (the ML function keeps the same name). The generated 4900 lines of ML contain the model-checker and all its prerequisites, like the code for red-black trees and other data structures.

For example, the main theorem proving the correctness of *cava*, the abstract function of *cava-code*, looks as follows:

theorem *cava-correct:*
$$\textit{cava bpc } \phi \leq \textbf{spec } \textit{res. res} = \textit{NO-LASSO} \leftrightarrow (\forall w.\ \textit{BP-accept bpc } w \longrightarrow w \models \phi)$$

It says that the *res*ult of *cava bpc* ϕ (where *bpc* is a compiled Boolean program, see above) satisfies the given **spec**ification: The result is *NO-LASSO* iff $\forall w.\ \textit{BP-accept bpc } w \longrightarrow w \models \phi$. See Section 5 for details on $\leq$ and **spec**. Here, w is an infinite sequence of (i. e. a function from $\mathbb{N}$ to) sets of Boolean variables of the program. The formula *BP-accept bpc* w is true iff there is a run of the program *bpc* such that at time point n exactly the variables $w(n)$ are true. Hence $\forall w \ldots$ states that every program run satisfies ϕ. The following lemma proves the refinement step:

lemma *cava-code-refine:*
$$\textbf{return } (\textit{cava-code bpc } \phi) \leq \textit{cava bpc } \phi$$

Once *cava-correct* and *cava-code-refine* have been proven, we can (almost automatically) prove the correctness of *cava-code*:

lemma *cava-code-correct:*
$$\textit{cava-code bpc } \phi = \textit{NO-LASSO} \leftrightarrow (\forall w.\ \textit{BP-accept bpc } w \longrightarrow w \models \phi)$$

4 A Closer Look at the Model Checker

This section describes and assembles the model checker components on the abstract level. At the end we summarize the size of the complete development.

4.1 Modeling Language

Our Boolean programs are similar to Dijkstra's guarded command programs, with all variables ranging over Booleans. There is `SKIP`, simultaneous assignment $v_1, \ldots, v_n$ `:=` $b_1, \ldots, b_n$ (where the b_i are Boolean expressions), sequential composition c_1 `;` c_2, conditional statements `IF [`$(b_1, c_1), \ldots, (b_n, c_n)$`] FI` (please excuse the syntax), and loops `WHILE` b `DO` c. We use (terminating) recursion in HOL to define programs that depend on some parameter. For example, the program for the n dining philosophers is defined via a function $dining(n)$ that returns a list of pairs of Boolean expressions and commands: $dining(0) = []$ and $dining(n+1) = \ldots\ dining(n) \ldots$. The overall program is simply `WHILE TT DO IF` $dining(n)$ `FI` (where `TT` is the constant true).

Since our focus is the model checker, we have refrained from extending our modeling language further, say with arrays or bounded integers. This would be straightforward in a higher-order interactive theorem prover, as the formalizations of much more complicated languages like C [17] and Java [12] have shown.

The semantics of our modeling language is formalized in HOL by a translation into a simple interpreted assembly language. The reason is speed: executing commands on the source code level is slow. Because the execution has to be interleaved with the state space exploration this would slow down the model checker considerably. The semantics of an assembly language program is given by a function *nexts* that computes the list of possible next configurations from a given configuration. Function *nexts* always returns a nonempty list (in the worst case by cycling), which means that every program has a run and all runs are infinite. Based on *nexts*, *BP-accept* is defined just as sketched at the end of Section 3 above.

4.2 LTL-to-Büchi Translator

The LTL-to-Büchi translator has two parts. The first part implements the algorithm of Gerth et al. [7] to translate an LTL formula into a generalized Büchi-automaton (LGBA, named *LGBArel* in the theories due to its transition *relation*). Recall that the acceptance condition of a generalized Büchi automaton consists of a set $\{F_0, \ldots, F_{m-1}\}$ of sets of accepting states. A run is accepting if it visits each F_i infinitely often. (Ordinary) Büchi automata are the special case $m = 1$. The function *LTL-to-LGBArel* (not shown) implements the tableau construction by Gerth et al. [7]. The correctness proof shows that the resulting LGBA recognizes the language of computations satisfied by the formula.

lemma *LTL-to-LGBArel-sound:*
LTL-to-LGBArel $\phi \leq$ **spec** A_L. $\forall w$. *LGBArel-accept* $A_L\ w \leftrightarrow w \models \phi$

Since the nested depth-first search algorithm only works for Büchi automata, not for generalized ones, the second part transforms LGBAs into equivalent Büchi automata. The construction for this is simple and well-known (see e. g. [4]), but we use this function to illustrate some points of our approach and (in the next section) of our use of the Refinement Framework.

We briefly recall the LGBA-to-Büchi construction. For each state q of the LGBA we have states (q, k) in the Büchi automaton, where $0 \leq k \leq m-1$ and m is the number of acceptance sets. If $q \to q'$ is a transition of the LGBA and q is labeled with a, then we add transitions $(q,k) \xrightarrow{a} (q',k)$ for every k to the Büchi automaton, but if q belongs to the i-th acceptance set then instead of $(q,i) \xrightarrow{a} (q',i)$ we add $(q,i) \xrightarrow{a} (q', i+1 \mod m)$. An additional technical point is that the LGBA produced by [7] carries labels on the states, and not on the transitions. Therefore, a label on a state of an LGBA must be translated into a label on all outgoing transitions of the corresponding Büchi automaton. Our abstract function for this is *LGBArel-to-BA*, shown below. The function takes an LGBA as an argument and returns a Büchi automaton with transition function *BA-Δ*, initial states *BA-I*, and a predicate *BA-F* defining the accepting states. The predicate *L A q a* is true if the state q of A is labeled by a.

```
LGBArel-to-BA A = do {
  Flist ← spec xs. F_GBA(A) = set xs ∧ distinct xs;
  return
   (| BA-Δ =
      (λ(q, k) a. if L A q a then
        {q' | (q,q') ∈ Δ_GBA(A)}
                  × {if k < length Flist ∧ q ∈ Flist!k
                     then (k+1) mod (length Flist)
                     else k}
                else {},
    BA-I = I_GBA(A) × {0},
    BA-F = (λ(q, k). (k = 0) ∧ (length Flist = 0 ∨ q ∈ Flist!0)) |) }
```

The acceptance family of A is a set of sets of states. However, for indexing, we actually need a *list* of sets. The second line of the above definition assigns this list to the name *Flist*. We prove language equivalence of the LGBA and its corresponding Büchi automaton:

```
lemma LGBArel-to-BA-sound: LGBA A_L ⟶
  LGBArel-to-Buchi A_L ≤
    spec A_B. ∀w. LGBArel-accept A_L w ↔ BA-accept A_B w
```

Assumption *LGBArel* A_L restricts the LGBA to fulfill some consistency properties. Function *LTL-to-BA* is the composition of *LTL-to-LGBArel* and *LGBArel-to-BA*. Combining their correctness lemmas yields the overall correctness lemma:

```
lemma LTL-to-BA-sound:
  LTL-to-BA φ ≤ spec A_B. ∀w. BA-accept A_B w ↔ w ⊨ φ
```

4.3 Language Emptiness Check

The model checker checks emptiness of the language of the product automaton using the algorithm of [22]. It is implemented in the function *nested-dfs-code*,

whose corresponding abstract function is *nested-dfs*. It is defined on any graph consisting of an initial vertex x, a successor function *succs*, and a distinguished subset F of vertices. In case of the product automaton, these correspond to the initial state, the transition function, and the set of accepting states of the automaton, respectively. The correctness theorem is phrased in terms of the transition relation $\rightarrow = \{(a,b) \mid b \in succs\ a\}$:

lemma *nested-dfs-NO-LASSO-iff:*
$nested\text{-}dfs\ (succs,\ F)\ x \leq$ **spec** $res.\ res = NO\text{-}LASSO \leftrightarrow$
$\neg(\exists\ v.\ x \rightarrow^* v \wedge v \in F \wedge v \rightarrow^+ v)$

It expresses that the function returns *NO-LASSO* iff there is no final state v reachable from x and reachable (in at least one step) from itself.

4.4 The Model Checker

Now we combine the individual components of the model checker to obtain the main function *cava*. We do not show this in the paper (see `http://cava.in.tum.de/CAV13`) but sketch (we do not explain all the details) how the individual correctness lemmas are combined into the main theorem *cava-correct* above.

In the first step, the input formula ϕ is translated into a Büchi automaton by *LTL-to-BA*. Lemma *LTL-to-BA-sound* states the correctness of this step. Then function *SA-BA-product* (not shown because straightforward) computes the product of the model A_S (which is an automaton-view of the program *bpc*) and the result A_B of the formula translation. The following lemma tells us that on the language level this corresponds to intersection:

lemma *SA-BA-product-correct:*
$SA\ A_S \wedge finite(BA\text{-}Q\ A_B)$
$\longrightarrow \mathcal{L}_{BA}\ (SA\text{-}BA\text{-}product\ A_S\ A_B) = \mathcal{L}_{BA}\ A_S \cap \mathcal{L}_{BA}\ A_B$

Note that $\mathcal{L}_{BA}\ A$ is simply the set of all w such that $BA\text{-}accept\ A\ w$. Now we characterize non-emptiness of the product automaton by the existence of a lasso, i. e. a path from a start state to an accepting state q_f together with a non-empty loop from q_f to q_f. The following lemma states the more interesting of the two implications:

lemma *Buchi-accept-lasso:*
$\mathcal{L}_{BA}\ A \neq \emptyset$
$\longrightarrow \exists q_i\ q_f\ r_1\ r_2.\ q_i \in BA\text{-}I\ A \wedge BA\text{-}F\ A\ q_f$
$\wedge\ is\text{-}finite\text{-}run\ A\ r_1 \wedge head\ r_1 = q_i \wedge last\ r_1 = q_f$
$\wedge\ is\text{-}finite\text{-}run\ A\ r_2 \wedge head\ r_2 = q_f \wedge last\ r_2 = q_f \wedge length\ r_2 > 1$

Combined with lemma *nested-dfs-NO-LASSO-iff* above (where $x \rightarrow^* v$ corresponds to $is\text{-}finite\text{-}run\ A\ r_1 \wedge head\ r_1 = x \wedge last\ r_1 = v$) this tells us that *nested-dfs* returns *NO-LASSO* iff the language is empty. Now it is just a set theoretic step that takes us to lemma *cava-correct* because *cava* builds the product of the model and the negated formula, which is empty iff every run of the program satisfies the formula.

4.5 Size of the Development

The following table summarizes the size of the development in lines of Isabelle "code", i. e. definitions and proofs, where typically 90% are proofs. We have distinguished the verification on the abstract level from the refinement steps:

	Abstract verification	Refinement
LTL-to-Büchi	3200	1500
Product construction	800	100
Emptiness check	2500	1600
Top level	500	400

In addition, there are approximately 5000 lines of supporting material that do not fit into the above classification. In total, this comes to roughly 16,000 lines. Moreover we rely on the separately developed and independent Collections Framework (30,000 lines, in total) and Refinement Framework (10,000 lines), described in Section 5.

5 Refinement Framework

When developing formally verified algorithms, there is a trade-off between the efficiency of the algorithm and the efficiency of the proof: For complex algorithms, a direct proof of an efficient implementation tends to get unmanageable, as proving implementation details blows up the proof and obfuscates the main ideas of the proof. A standard approach to this problem is stepwise refinement [2,3], where this problem is solved by modularization of the correctness proof: One starts with an abstract version of the algorithm and then refines it (in possibly many steps) to the concrete, efficient version. A refinement step may reduce the nondeterminism of a program and replace abstract datatypes by their implementations. For example, selection of an arbitrary element from a set may be refined to getting the head of a list. The main point is, that correctness properties can be transferred over refinements, such that correctness of the concrete program easily follows from correctness of the abstract algorithm and correctness of the refinement steps. The abstract algorithm is not cluttered with implementation details, such that its correctness proof can focus on the main algorithmic ideas. Moreover, the refinement proofs only focus on the local changes in a particular refinement step, not caring about the overall correctness property.

In Isabelle/HOL, refinement is supported by the Refinement Framework [15,16] and the Isabelle Collection Framework [14,13]. The former framework implements a refinement calculus [3] based on a nondeterminism monad [24], and the latter one provides a large collection of verified efficient data structures. Both frameworks come with tool support to simplify their usage for algorithm development and to automate canonical tasks such as verification condition generation.

In the nondeterminism monad, each program yields a *result* that is either a set of possible values or the special result **fail**. A result r *refines* another result r', written $r \leq r'$, if $r' = \textbf{fail}$ or if $r \neq \textbf{fail} \neq r'$ and every value of r is also

```
dfs E vd v0 = do {
  rec_T (λD (V,v).
    if v = vd then return True
    else if v ∈ V then return False
    else do {
      let V = insert v V;
      foreach_C {v' | (v,v') ∈ E} (λx. x=False)
          (λv' -. D (V,v')) False
    }
  ) ({},v0) }
```

Algorithm 1. Simple Depth-First-Search Algorithm

```
function dfs(E : set of pairs of 'a, vd : 'a, v0 : 'a)
    return dfs-body (E, vd, v0, ∅, v0)

function dfs-body(E, vd, v0, V : set of 'a, v : 'a)
    if v = vd then return true
    else if v ∈ V then return false
    else
        V := V ∪ {v}
        ret := false
        for all x ∈ {v' | (v, v') ∈ E} do
            if ret = true then break
            else ret := dfs-body (E, vd, v0, V, x)
        return ret
```

Algorithm 2. Simple DFS Algorithm (imperative equivalent)

a value of r'. The result **return** x contains the single value x, and the result **spec** x. Φ contains all values x that satisfy the predicate Φ. Thus, correctness of a program f w. r. t. precondition P and postcondition Q can be specified as: $P\ x \longrightarrow f\ x \leq$ **spec** r. Q. Intuitively, this reads as: If the argument x satisfies precondition P, then all possible result values of f satisfy postcondition Q.

As an example we present a depth-first search algorithm, which is a simplified version of the nested DFS algorithm used in the model checker. Algorithm 1 uses the syntax of Isabelle/HOL, and Algorithm 2 displays its equivalent in imperative pseudocode. In Isabelle/HOL, a Haskell-like **do**-notation is used. The $\mathbf{rec_T}$ combinator is recursion, where the recursive call is bound to the first parameter D. The $\mathbf{foreach_C}$ combinator iterates over all elements of the set, and additionally has a continuation condition, i. e. the iteration is terminated if the continuation condition does not hold any more. Here, we use the continuation condition to break the loop if the recursive call returns true.

We now prove the following lemma, stating that if the algorithm returns true, then the node *vd* is reachable from *v0*:

lemma *dfs-sound:*
finite $\{v.\ (v0,v) \in E^*\} \longrightarrow$ *dfs E vd v0* $\leq$ **spec** $r.\ r \longrightarrow (v0,vd) \in E^*$

The proof of this lemma in Isabelle/HOL reads as follows:

unfolding *dfs-def*
apply (*refine-rcg refine-vcg impI*
RECT-rule[**where**
$\Phi = \lambda(V,v).\ (v0,v) \in E^* \wedge V \subseteq \{v.\ (v0,v) \in E^*\}$ **and**
$V =$ *finite-psupset* $(\{v.\ (v0,v) \in E^*\}) \times_{\mathsf{lex}} \{\}$]
FOREACHc-rule[**where** $I = \lambda\text{-}\ r.\ r \longrightarrow (v0,\ vd) \in E^*$])
... [*3 lines of straightforward Isabelle script*]

In the first line, we unfold the definition of *dfs*. In the **apply**-command starting in the second line, we invoke the verification condition generator. The crucial part here is to specify the right invariants: For the recursion, we need a precondition Φ and a variant V. The precondition states that the current node v is reachable and that the set of visited nodes is reachable. The variant states that, in each recursive call, the set of visited nodes gets closer to the finite set of reachable nodes. This is required to show termination. For the foreach-loop, we need an invariant I. It states that, if we break the loop, the target node vd is reachable. The remaining lines of the proof show that the precondition implies the postcondition, that the variant is valid, and that the invariant is preserved. As this only involves argumentation about sets, which enjoy good tool support in Isabelle/HOL, this can be done in a few straightforward lines of Isabelle script.

Once we have defined the abstract algorithm and proved that it satisfies its specification, we refine it to an executable version. This includes *data refinement*, e. g. implementing sets by red-black trees, and nondeterminism reduction, e. g. implementing the foreach-loop by in-order iteration over the red-black tree.

A *refinement relation* is a single-valued relation between concrete and abstract values (e. g. between red-black trees and sets). Single-valuedness means that a concrete value must not be related to more than one abstract value.

The *concretization function* $\Downarrow$ lifts a refinement relation R to results. For programs f and f', the statement $(x,x') \in Ri \longrightarrow f\ x \leq \Downarrow Ro\ (f'\ x')$ means that program f refines program f', where the argument is refined according to relation Ri, and the result is refined according to the relation Ro.

The refinement of *dfs* is straightforward, and the Refinement Framework can generate an executable version together with the refinement proof automatically.

As an example for a more complex refinement, reconsider *LGBArel-to-Buchi* from Section 4.2, which translates node-labeled generalized Büchi automata to edge-labeled Büchi automata. Its refined version is Algorithm 3. Here, the input automaton is represented by a tuple, where the transitions D are represented by nested red-black trees, the sets of initial states I and final states F are represented by lists of distinct elements, and the representation of the labeling function L is not changed. The result automaton is represented by a tuple consisting of

- a successor function, which maps a state and a label to a list of distinct successor states,
- a list of distinct initial states, and
- the characteristic function of the sets of final states.

```
LGBArel-to-Buchi-impl ((-, -, D, I, F), L) = do {
  let Flist = lsi-to-list F;
  return
    (λ(q, k) l.
      if L q l then
        (let k' = (if (k < length Flist ∧ lsi-memb q (Flist!k))
                   then (k+1) mod (length Flist) else k) in
         let succs = rs-lts-succ-it D q () (λ-. True) lsi-ins (lsi-empty ()) in
           lsi-product succs (list-to-lsi [k']))
       else lsi-empty (),
      lsi-product I (list-to-lsi [0]),
      (λ(q, k). (k = 0) ∧ (length Flist = 0 ∨ lsi-memb q (Flist!0))))
}
```

Algorithm 3. Implementation of the LGBA-to-Büchi translation

For this refinement, the **spec**-statement, which was used to nondeterministically select a list representation of the set of final states, has been replaced by a let statement, which deterministically uses the list *lsi-to-list F*. In the **return**-statement, we have replaced the abstract operations on sets (e. g. $\times$) by their concrete counterparts (e. g. *lsi-product*). Note that functions from the Isabelle Collection Framework follow a standard naming scheme: The prefix *lsi* denotes operations on sets represented by lists of distinct elements. Analogously, the prefix *rs* denotes operations on sets represented by red-black trees.

The following lemma relates the abstract and the concrete algorithm:

lemma *LGBArel-to-BA-impl-refine:* $(A_L, A'_L) \in$ *LGBArel-impl-rel* $\longrightarrow$
LGBArel-to-BA-impl $A_L \leq \Downarrow$*BA-impl-rel* (*LGBArel-to-BA* A'_L)

Here, *LGBArel-impl-rel* relates the input automaton A'_L to its representation A_L. Similarly, *BA-impl-rel* relates the result automaton to its representation. The proof of the above lemma is quite straightforward. The main proof effort goes into showing that the successor states of q (abstractly: $\{q' \mid (q,q') \in \Delta\_GBA(A)\}$) are correctly implemented by the iterator *rs-lts-succ-it*, which iterates over the successor states and collects them in a list.

The algorithm *LGBArel-to-BA-impl* is already deterministic. However, it is still defined in the nondeterminism monad, which is not executable. Thus, a further refinement step removes the nondeterminism monad. This step is fully automatic: The Refinement Framework defines a constant *LGBArel-to-BA-code* and proves the lemma **return** (*LGBArel-to-BA-code* A_L) $\leq$ *LGBArel-to-BA-impl* A_L. Finally, the code-generator exports ML-code for *LGBArel-to-BA-code*.

Using transitivity of $\leq$ and monotonicity of the concretization function, we could combine the above result with Lemma *LGBArel-to-BA-sound* from Section 4.2, and obtain:

lemma *LGBArel-to-BA-code-sound:* $(A_L, A'_L) \in$ *LGBArel-impl-rel* $\longrightarrow$
$\exists A_B.$ (*LGBArel-to-BA-code* A_L,A_B) $\in$ *BA-impl-rel*
$\wedge$ ($\forall w.$ *LGBArel-accept* A'_L $w \leftrightarrow$ *BA-accept* A_B w)

Note, however, that we need to prove such lemmas only for the interface functions of our tool, not for internal functions like *LGBArel-to-BA*.

6 Some Experiments

As mentioned in the introduction, our project must fulfill two conflicting requirements: a mechanized proof of functional correctness, and adequate performance for a reference implementation. In this section we provide some evidence for the latter.

The natural tool for a comparison with our checker is SPIN [10], while keeping in mind that SPIN is implemented in C, while our checker is implemented in ML. We use SPIN version 6.2.3 with turned-off optimizations (`-o1 -o2 -o3 / -DNOREDUCE`), and MLton version 20100608 as the compiler for our checker. We take three standard well-known and easily scalable benchmarks: the Dining Philosophers, a Readers-Writers system guaranteeing concurrent read but exclusive write, and the Leader-Filters example of [5].

The comparison with SPIN requires some care because the compilation of a program into a Kripke structure can lead to substantial differences in the number of reachable states. For instance, consider a program $P_1 \parallel \ldots \parallel P_n$, where $P_i =$ **while true do** $b := 0$ (in a generic program notation). Each parallel component can be represented by an automaton with one single state and a self-loop labeled by $b := 0$. If initially $b = 0$, then the complete system has one reachable state. However, a compilation process might also lead to an automaton that cycles between two states, moving from the first to the second state by means of a transition labeled by $b := 0$, and back to the initial state by a silent transition. This harmless change has a large impact in the state space: the new version has 2^n reachable states, where n is the number of components, leading to much larger verification times. We have observed this effect in our experiments: SPIN's mature compilation process generates fewer states than our tool, where this aspect has not yet been optimized (cf. Table 1).[1]

For this reason, we compare not only the time required to explore the state space of the product automaton, but also the *state exploration speed*, i. e. the number of states explored per time unit.

If a property does not hold, then the verificiation time and the number of states explored may depend on arbitrary choices in how the depth-first search is conducted. So we only consider properties that hold, for which every tool explores the complete state space. Table 1 shows verification times for the trivial property $\mathbf{G}\mathit{true}$. All times are in milliseconds. Since experiments with other properties yield similar results and no new insight, the results are omitted. We observe that our checker is between 7 and 26 times slower than SPIN.

We now consider the exploration speed. Since it depends on the number of states (if the number is large then state descriptors are also large, and more costly

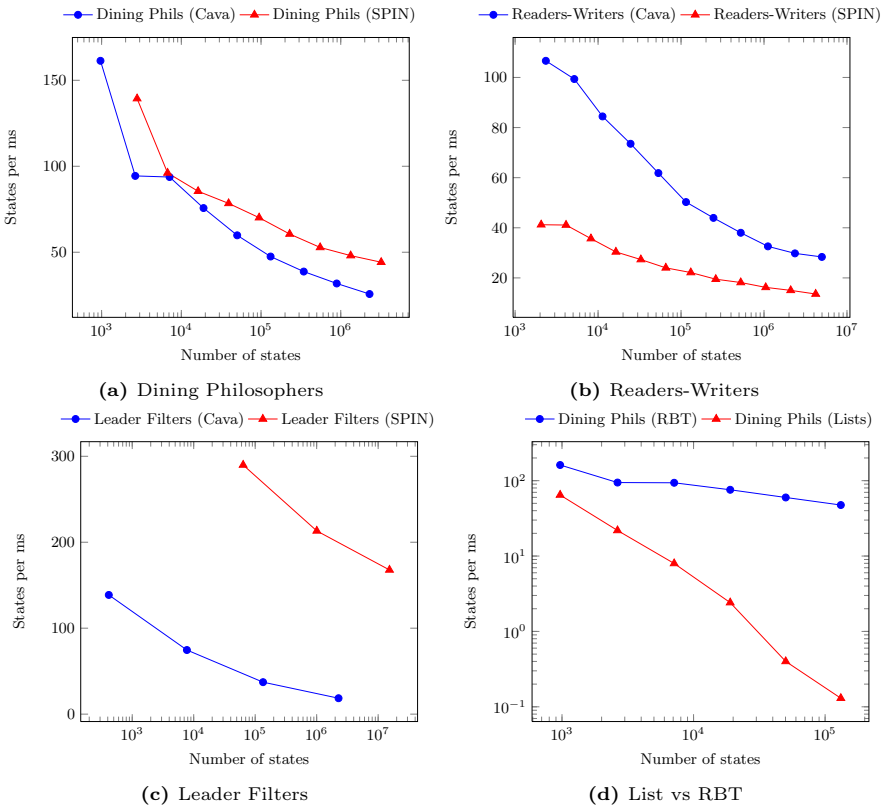
[1] A similar effect is observed in the number of states of the Büchi automaton for a formula: While both SPIN and our checker are based on the algorithm of [7], it is known that the number of states is very sensitive to simplification heuristics.

Table 1. Construction time (ms) for the state space (in thousands of states)

Phils					RW					LF				
	SPIN		Cava			SPIN		Cava			SPIN		Cava	
#	Time	States	Time	States	#	Time	States	Time	States	#	Time	States	Time	States
10	70	6	839	50	10	20	1	134	11	3	10	4	104	8
11	190	16	2773	132	11	50	2	335	25	4	220	64	3611	134
12	500	39	8857	343	12	100	4	862	53	5	4720	1006	122620	2271
13	1350	95	27957	890	13	230	8	2283	115	6	91200	15305	OoM	

to process), we plot it against the size of the state space. Overall, our checker generates about 10^4–10^5 states per second (this was also the speed reported in [10], published in 2003), and is consistent with the fact that after one decade SPIN is about one order of magnitude faster. Figures 1a–1c show the results for each of the three benchmarks, with exploration speed in states per *millisecond*. Our checker is about 3 times faster than SPIN on *Readers-Writers*, about eight times slower on *Leader Filters*, and about as fast as SPIN on *Dining Philosophers*.

Summarizing, while our checker is slower than SPIN, we think it is fast enough for the purpose of a reference implementation. Most of the functionality of an



(a) Dining Philosophers **(b)** Readers-Writers

(c) Leader Filters **(d)** List vs RBT

Fig. 1. State-exploration speed

LTL model checker can be tested on examples with 10^4–10^7 states, for which our checker is about one order of magnitude slower than SPIN, which will be somewhat reduced once our compilation process is optimized.

We can now also illustrate the importance of the Refinement Framework. Without it, we would at most have been able to prove correctness of a model checker with sets implemented as lists. Using the Refinement Framework we can easily generate code for this "slow" checker, and compare its speed with the one of the optimized version where sets are implemented as red-black trees. The result for the dining philosophers is shown in Figure 1d, which uses a double logarithmic scale. For systems with 10^5 states the slow checker is already almost three orders of magnitude slower, which makes it fully inadequate as a reference implementation.

7 Conclusion

Model checkers are a paradigm case of systems for which both correctness and efficiency are absolutely crucial. We have presented the—to the best of our knowledge—first model checker whose code has been fully verified using a theorem prover and is efficient enough to constitute a reference implementation for testing purposes. A key element was our use of a refinement process: specify and verify the model checker on an abstract mathematical level (about 250 lines of code, not counting comments etc.), then improve efficiency of the algorithms and data structures by stepwise refinement, and finally let the theorem prover generate ML code (4900 lines). Our experiments indicate that our checker is only one order of magnitude slower than SPIN in the range of systems with 10^6–10^7 states. We think this result is very satisfactory, since SPIN is a highly optimized checker, programmed in C; moreover, our results indicate that the distance to SPIN can be shortened by means of optimizations in the compiler that generates the state space from the high-level system model.

An alternative approach to obtain verified results is to use checkers that provide certificates of their answer (e. g. a Hoare proof) that can be independently checked by a trusted certifier [18,20]. The advantage of this approach is a much smaller formalization effort, and its disadvantage the potentially very large size of the certificates (worst case: the same order of magnitude as the state space).

References

1. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (1996)
2. Back, R.J.: On the correctness of refinement steps in program development. Ph.D. thesis, Department of Computer Science, University of Helsinki (1978)
3. Back, R.J., von Wright, J.: Refinement Calculus — A Systematic Introduction. Springer (1998)
4. Baier, C., Katoen, J.P.: Principles of Model Checking. MIT Press (2008)
5. Choy, M., Singh, A.K.: Adaptive solutions to the mutual exclusion problem. Distributed Computing 8(1), 1–17 (1994)
6. Courcoubetis, C., Vardi, M., Wolper, P., Yannakakis, M.: Memory-efficient algorithms for the verification of temporal properties. Formal Methods in System Design 1(2/3), 275–288 (1992)

7. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: Dembinski, P., Sredniawa, M. (eds.) Proc. Int. Symp. Protocol Specification, Testing, and Verification. IFIP Conference Proceedings, vol. 38, pp. 3–18. Chapman & Hall (1996)
8. Haftmann, F., Nipkow, T.: Code generation via higher-order rewrite systems. In: Blume, M., Kobayashi, N., Vidal, G. (eds.) FLOPS 2010. LNCS, vol. 6009, pp. 103–117. Springer, Heidelberg (2010)
9. Holzmann, G., Peled, D., Yannakakis, M.: On nested depth first search. In: Grégoire, J.C., Holzmann, G.J., Peled, D.A. (eds.) Proc. of the 2nd SPIN Workshop. Discrete Mathematics and Theoretical Computer Science, vol. 32, pp. 23–32. American Mathematical Society (1997)
10. Holzmann, G.J.: The Spin Model Checker — Primer and Reference Manual. Addison-Wesley (2003)
11. Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., Winwood, S.: seL4: formal verification of an OS kernel. In: Matthews, J.N., Anderson, T.E. (eds.) Proc. ACM Symp. Operating Systems Principles, pp. 207–220. ACM (2009)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine and compiler. ACM Trans. Progr. Lang. Syst. 28(4), 619–695 (2006)
13. Lammich, P., Lochbihler, A.: The Isabelle Collections Framework. In: Kaufmann, M., Paulson, L.C. (eds.) ITP 2010. LNCS, vol. 6172, pp. 339–354. Springer, Heidelberg (2010)
14. Lammich, P.: Collections framework. In: Archive of Formal Proofs (December 2009), formal proof development, `http://afp.sf.net/entries/Collections.shtml`
15. Lammich, P.: Refinement for monadic programs. In: Archive of Formal Proofs (2012), formal proof development, `http://afp.sf.net/entries/Refine_Monadic.shtml`
16. Lammich, P., Tuerk, T.: Applying data refinement for monadic programs to Hopcroft's algorithm. In: Beringer, L., Felty, A. (eds.) ITP 2012. LNCS, vol. 7406, pp. 166–182. Springer, Heidelberg (2012)
17. Leroy, X.: A formally verified compiler back-end. J. Automated Reasoning 43, 363–446 (2009)
18. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
19. Nipkow, T., Paulson, L., Wenzel, M.: Isabelle/HOL — A Proof Assistant for Higher-Order Logic. LNCS, vol. 2283. Springer (2002)
20. Peled, D., Pnueli, A., Zuck, L.D.: From falsification to verification. In: Hariharan, R., Mukund, M., Vinay, V. (eds.) FSTTCS 2001. LNCS, vol. 2245, pp. 292–304. Springer, Heidelberg (2001)
21. Schimpf, A., Merz, S., Smaus, J.-G.: Construction of Büchi automata for LTL model checking verified in Isabelle/HOL. In: Berghofer, S., Nipkow, T., Urban, C., Wenzel, M. (eds.) TPHOLs 2009. LNCS, vol. 5674, pp. 424–439. Springer, Heidelberg (2009)
22. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
23. Sprenger, C.: A verified model checker for the modal μ-calculus in Coq. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 167–183. Springer, Heidelberg (1998)
24. Wadler, P.: Comprehending monads. Mathematical Structures in Computer Science 2, 461–478 (1992)

Automatic Generation of Quality Specifications

Shaull Almagor, Guy Avni, and Orna Kupferman

School of Computer Science and Engineering, The Hebrew University, Jerusalem, Israel

Abstract. The logic $\mathrm{LTL}^{\triangledown}$ extends LTL by quality operators. The satisfaction value of an $\mathrm{LTL}^{\triangledown}$ formula in a computation refines the $0/1$ value of LTL formulas to a real value in $[0,1]$. The higher the value is, the better is the quality of the computation. The quality operator $\triangledown_\lambda$, for a quality constant $\lambda \in [0,1]$, enables the designer to prioritize different satisfaction possibilities. Formally, the satisfaction value of a sub-formula $\triangledown_\lambda \varphi$ is λ times the satisfaction value of φ. For example, the $\mathrm{LTL}^{\triangledown}$ formula $G(req \rightarrow (X\,grant \vee \triangledown_{\frac{1}{2}} F\,grant))$ has value 1 in computations in which every request is immediately followed by a grant, value $\frac{1}{2}$ if grants to some requests involve a delay, and value 0 if some request is not followed by a grant.

The design of an $\mathrm{LTL}^{\triangledown}$ formula typically starts with an LTL formula on top of which the designer adds the parameterized $\triangledown$ operators. In the Boolean setting, the problem of automatic generation of specifications from binary-tagged computations is of great importance and is a very challenging one. Here we consider the quantitative counterpart: an $\mathrm{LTL}^{\triangledown}$ query is an $\mathrm{LTL}^{\triangledown}$ formula in which some of the quality constants are replaced by variables. Given an $\mathrm{LTL}^{\triangledown}$ query and a set of computations tagged by satisfaction values, the goal is to find an assignment to the variables in the query so that the obtained $\mathrm{LTL}^{\triangledown}$ formula has the given satisfaction values, or, if this is impossible, best approximates them. The motivation to solving $\mathrm{LTL}^{\triangledown}$ queries is that in practice it is easier for a designer to provide desired satisfaction values in representative computations than to come up with quality constants that capture his intuition of good and bad quality.

We study the problem of solving $\mathrm{LTL}^{\triangledown}$ queries and show that while the problem is NP-hard, interesting fragments can be solved in polynomial time. One such fragment is the case of a single tagged computation, which we use for introducing a heuristic for the general case. The polynomial solution is based on an analysis of the search space, showing that reasoning about the infinitely many possible assignments can proceed by reasoning about their partition into finitely many classes. Our experimental results show the effectiveness and favorable outcome of the heuristic.

1 Introduction

Traditional formal methods are based on a Boolean satisfaction notion – a reactive system satisfies, or not, a given specification. In recent years there is growing need and interest in formalizing and reasoning about quantitative systems and properties. This includes, for example, probabilistic [16], fuzzy [18], and accumulative [9] settings. An exciting direction in this effort is the development of formalisms and methods for reasoning about the *quality* of systems [1,2]. The working assumption in these works is that

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 479–494, 2013.

satisfying a specification is not a yes/no matter. Different ways of satisfying a specification should induce different levels of quality, which should be reflected in the semantics of the specification formalism. In particular, in [2], the authors introduce an extension of linear temporal logic (LTL [19]) by a quantitative layer that enables the designer to prioritize different satisfaction possibilities. In the extended setting, the satisfaction value of a formula in a computation refines the $0/1$ value of LTL formulas to a real value in $[0, 1]$. The higher the value is, the better is the quality of the computation.

The extension uses a family of *propositional quality operators*. A basic such operator is $\triangledown_\lambda$, for a *quality constant* $\lambda \in [0, 1]$ that multiplies the satisfaction value of its operand by λ. We consider the logic $\mathrm{LTL}^\triangledown$, which extends LTL by the $\triangledown_\lambda$ operator. The standard LTL operators are adjusted in $\mathrm{LTL}^\triangledown$ to values in $[0, 1]$: disjunctions are interpreted as $\max$, negation as subtraction from 1, and so on. For example, the satisfaction value of the formula $\psi_1 \vee \triangledown_{\frac{1}{2}} \psi_2$ in a computation π is the maximum between the satisfaction value of ψ_1 in π, and $\frac{1}{2}$ the satisfaction value of ψ_2 in π. As a more elaborate example, consider a system that grants locks to a data structure. The system can grant either a read-only lock or a read-write lock. The quality of the system may be specified as $\mathsf{G}(\mathit{req} \rightarrow \mathsf{X}(\mathit{read\text{-}write} \vee (\triangledown_{\frac{3}{4}} \mathit{read\text{-}only}))$, stating that receiving read-write lock gives satisfaction value of 1, whereas receiving a read-only lock reduces that satisfaction value to $\frac{3}{4}$, In [2], the authors demonstrate the usefulness of the ability to specify quality and solve the model-checking and synthesis problems for $\mathrm{LTL}^\triangledown$.

Already in the Boolean setting, both model checking and synthesis rely on the specification to accurately reflect the designer's intention. One of the criticisms against formal method is that the latter challenge, of coming up with correct specifications, is not much easier than model checking or synthesis. Thus, formal methods merely shift the difficulty of developing correct implementations to that of developing correct specifications [13]. *Property assurance* is the activity of eliciting specifications that faithfully capture designer intent [8,21]. One approach for property assurance is to challenge given specifications with sanity checks like non-validity, satisfiability, and vacuity [15]. More involved quality checks are studied in the PROSYD project [20] [1]. A second approach is that of automatic generation of specifications. This includes ideas from *learning*, where specifications given by means of automata are learned from a sample of behaviors tagged as good or bad [6,17], methods based on a generation of specifications from basic *patterns* [11], and *specification mining*, where specifications are generated by analyzing the runs of the given system [4]. In the novel quantitative setting, there is (yet) no experience in specification design nor tools or methods for property assurance. In this paper, we introduce such a method and study the problem of automatically generating the quality layer in $\mathrm{LTL}^\triangledown$ formulas.

The design of an $\mathrm{LTL}^\triangledown$ formula typically starts with an LTL formula on top of which the designer adds the parameterized $\triangledown$ operators. The underlying assumption behind our approach is that in practice it is easier for a designer to provide desired satisfaction values in representative computations than to come up with quality constants that capture his intuition of high and low quality. This resembles the classical process of learning

[1] A related line of research is that of specification debugging [5], where, in the process of model checking, counterexamples are automatically clustered together in order to make the manual debugging of temporal properties easier.

a hypothesis from tagged samples. Formally, an LTL^{∇} *query* is an LTL^{∇} formula in which some of the quality constants are replaced by variables. A *path constraint* is a pair $\langle \pi, I \rangle$, where π is a lasso-shaped path and $I \subseteq [0,1]$ is a closed interval. Consider an LTL^{∇} query φ with variables in $\mathcal{X}$. For an assignment $f : \mathcal{X} \to [0,1]$, we use φ^f to denote the LTL^{∇} formula obtained from φ by replacing each variable $x \in \mathcal{X}$ by $f(x)$. The LTL^{∇} query problem is to find, given an LTL^{∇} query φ and a set $\mathcal{C}$ of path constraints, an assignment f to the variables in φ so that φ^f satisfies all the constraints (or returns that no such assignment exists). Thus, for all $\langle \pi, I \rangle \in C$, the satisfaction value of φ^f in π is in the interval I. Note that I may (but need not) be a single point. Note that beyond the restrictions on the quality constants in φ^f that follow from the constraints, restrictions may be induced also by repeated occurrences of the same variable. Subtle connections between different constants can be specified too, using nesting. In practice, however, most queries are simple (that is, each variable appears only once) and are free of nesting.

As an example, consider the following specification: "After a request, an ack should ideally be given immediately and hold for two time units. An ack that holds only for one time unit is also acceptable, provided that it is given within two time units". A designer that wants to formalizes "ideally" and "acceptable" may have a clear idea that he wants to upper bound the satisfaction value of a policy with a single time-unit ack by $\frac{3}{4}$ but may find it difficult to come up with the exact "penalty" for a delay in this case. This situation is captured by the following LTL^{∇} query:

$$\mathsf{G}(req \rightarrow ((\mathsf{X}ack \wedge \mathsf{XX}ack) \vee \nabla_{\frac{3}{4}}(\nabla_x(\mathsf{X}ack) \vee \nabla_y(\mathsf{XX}ack)))).$$

The satisfaction value of an induced LTL^{∇} formula in a computation with an ideal ack policy is 1. In a computation with a single time-unit ack, it is at most $3/4$, to be further tuned down by the assignments to x and y. Grading given behaviors is easier than finding an assignment that captures the designer's intuition. For example, the designer may declare that a computation $(\{req\}, \{ack\})^\omega$ is not that bad, and satisfies the specification with quality $\frac{3}{4}$. Also, the path $(\{req\}, \emptyset, \{ack\})^\omega$ satisfies it with quality in $[\frac{1}{3}, \frac{1}{2}]$. A solution to the corresponding LTL^{∇} query problem suggests an assignment to x and y that satisfies the designer's constraints. For example, $x = 1$ and $y = \frac{1}{2}$. As we discuss in Section 2.2, it is possible to automate not only the generation of quality constraints, but also the generation of LTL^{∇} queries out of LTL formulas.

Before we continue to describe our results, let us review other settings with partially-specified systems or specifications. In the Boolean setting, reasoning about partially-specified systems is useful in automatic partial synthesis [22] and program repair [14]. From the other direction, partially-specified specifications are used for system exploration. In particular, in *query checking* [10], the specification contains variables, and the goal is to find an assignment to the variables with which the explored system satisfies the specification. While the formulation of the problem is similar, the motivation is very different, as the goal is to explore, synthesize, or reason about the system, whereas our goal here is automatic generation of specifications. The fact we consider the quantitative setting makes the underlying considerations and algorithms very different too. In the quantitative setting, related work includes parameterized weighted containment [7], where a partially specified weighted automaton is given, and the goal is to find an assignment to the missing weights such that a containment constraint is met.

An orthogonal research direction is that of parametric real-time reasoning [3]. There, the quantitative nature of the automata origins from real-time constraints, the semantics is very different, and the goal is to find restrictions on the behavior of the clocks such that the automata satisfy certain properties.

We start by showing that in general, the $\mathrm{LTL}^{\triangledown}$ query problem is NP-hard. Checking whether a suggested assignment satisfies the set of constraints can be done in polynomial time, suggesting that the problem is in NP. One, however, also has to consider the domain and representation of the interval constraints and the assignment. For example, the only solution to the query $\triangledown_x \triangledown_x p$ and the constraint $\langle \{p\}^\omega, \frac{1}{2} \rangle$ assigns to x the value $1/\sqrt{2}$, which is irrational and thus does not have a finite representation in a binary expansion. For common queries, in particular simple queries without nesting of variables, we are able to prove that a "short" satisfying assignment exists, making the problem NP-complete.

We proceed to study a fragment of the problem, where the $\mathrm{LTL}^{\triangledown}$ queries are simple and the set of constraints includes a single computation. We show that in this case, the problem can be solved in polynomial time.[2] Our polynomial algorithm is based on an analysis of the search space, showing that the infinitely many possible assignments to each of the variables x in the query can be rounded up to linearly many ones, depending on the desired satisfaction value and the structure of the formula inside which x is nested. The induced space of possible assignments can be then searched efficiently.

Finally, we use the case of a single constraint in a heuristic for the general case. To do so, it is convenient to consider the problem in a geometrical perspective: consider an $\mathrm{LTL}^{\triangledown}$ query φ over k variables and a set of constraints $\mathcal{C}$. Every constraint $\langle \pi, I \rangle \in \mathcal{C}$ induces a set $S_{\langle \pi, I \rangle, \varphi} \subseteq [0,1]^k$ of solutions to that constraint. The $\mathrm{LTL}^{\triangledown}$ query problem for $\mathcal{C}$ amounts to finding a point $f \in \bigcap_{\langle \pi, I \rangle \in \mathcal{C}} S_{\langle \pi, I \rangle, \varphi}$. [3] The geometrical perspective makes it easy to define an optimization problem: we seek an assignment that minimizes the sum (over $\langle \pi, I \rangle \in \mathcal{C}$) of distances to $S_{\langle \pi, I \rangle, \varphi}$. We suggest three heuristics for finding an assignment, and evaluate them according to two *loss function*, namely ways to define distances in the $[0,1]^k$ space: number of constraints satisfied, and distance in $\| \cdot \|_2$. The three heuristics combine a consideration of external assignments as well as the center of gravity of the assignments for the underlying constraints. We implemented our algorithms and examined the quality constants generated for various specifications. As detailed in Section 6, our results show the effectiveness and favorable outcome of the approach and algorithms.

Due to lack of space, most proofs are omitted and can be found in the full version at the authors' URLs.

2 The Logic $\mathrm{LTL}^{\triangledown}$

The logic $\mathrm{LTL}^{\triangledown}$ is a multi-valued logic that extends the linear temporal logic LTL with a parameterized quality operator ∇_λ. The logic, along with model-checking and

[2] We note that both requirements, of simple queries and a singleton constraint are needed; removing one of them we are back to NP-hardness.

[3] The main difficulty in solving the $\mathrm{LTL}^{\triangledown}$ query problem is that even for a single constraint, this set may not be convex, and therefore not amenable to methods of convex analysis.

synthesis algorithms for it, was introduced in [2]. We start by defining its syntax and semantics. Let AP be a set of Boolean atomic propositions[4]. An LTL$^{\nabla}$ formula is one of the following:

- $\texttt{True}$, $\texttt{False}$, or p, for $p \in AP$.
- $\neg\varphi$, $\varphi \vee \psi$, $\nabla_\lambda \varphi$, $\mathsf{X}\varphi$, or $\varphi \mathsf{U} \psi$, for LTL$^{\nabla}$ formulas φ and ψ, and a *quality constant* $\lambda \in [0,1]$.

The semantics of LTL$^{\nabla}$ is defined with respect to infinite computations over AP. Each position in the computation corresponds to a valuation to the atomic propositions, thus a computation is a word $\pi = \pi_0, \pi_1, \ldots \in (2^{AP})^\omega$. We use π^i to denote the suffix $\pi_i, \pi_{i+1}, \ldots$ of π. The semantics maps a computation π and an LTL$^{\nabla}$ formula φ to the satisfaction value of φ in π, denoted $[\![\pi, \varphi]\!]$. The satisfaction value is in $[0,1]$, defined by induction on the structure of φ as described in Table 1 below. As with LTL, we use $\mathsf{F}\psi$ ("eventually") and $\mathsf{G}\psi$ ("always") as abbreviations for $\texttt{True}\mathsf{U}\psi$ and $\neg\mathsf{F}\neg\psi$, respectively, as well as the standard Boolean abbreviations $\wedge$ and $\rightarrow$.

Table 1. The semantics of LTL$^{\nabla}$

Formula	Satisfaction Value
$[\![\pi, \texttt{True}]\!]$	1
$[\![\pi, \texttt{False}]\!]$	0
$[\![\pi, p]\!]$	1 if $p \in \pi_0$ 0 if $p \notin \pi_0$
$[\![\pi, \neg\varphi]\!]$	$1 - [\![\pi, \varphi]\!]$
$[\![\pi, \varphi \vee \psi]\!]$	$\max([\![\pi, \varphi]\!], [\![\pi, \psi]\!])$
$[\![\pi, \nabla_\lambda \varphi]\!]$	$\lambda \cdot [\![\pi, \varphi]\!]$
$[\![\pi, \mathsf{X}\varphi]\!]$	$[\![\pi^1, \varphi]\!]$
$[\![\pi, \varphi \mathsf{U} \psi]\!]$	$\max_{i \geq 0}\{\min\{[\![\pi^i, \psi]\!], \min_{0 \leq j < i} [\![\pi^j, \varphi]\!]\}\}$

Evaluating LTL$^{\nabla}$ Formulas on Lasso Computations. We say that a computation π is a *lasso* if $\pi = u \cdot v^\omega$, for finite computations $u, v \in (2^{AP})^*$ with $v \neq \epsilon$. We refer to u as the *prefix* of the lasso and to v as its *cycle*. The standard bottom-up labeling algorithm for model checking LTL formulas with respect to lasso computations can be easily extended to LTL$^{\nabla}$. The algorithm is based on the simple observation that if $[\![\pi^i, \psi]\!]$ is known for all $i \geq 0$ and subformulas ψ of φ, then it is possible to calculate, in time linear in $|u| + |v|$, the values $[\![\pi^i, \varphi]\!]$, for all $i \geq 0$. Indeed, the periodicity of π implies that there are only $|u| + |v|$ different suffixes to consider, and, by the semantics of LTL$^{\nabla}$, the satisfaction value of φ can be easily inferred from the satisfaction value of its subformulas. The only non-trivial case is when $\varphi = \psi_1 \mathsf{U} \psi_2$, but also there, one can start with the satisfaction value of ψ_2 and then repeatedly go back the lasso checking for every suffix whether, taking the satisfaction value of ψ_1 into an account, it is worthwhile to postpone the satisfaction of the eventuality. To conclude, we have the following.

Proposition 1. *Given an* LTL$^{\nabla}$ *formula* φ *and finite computations* $u, v \in (2^{AP})^*$*, calculating* $[\![u \cdot v^\omega, \varphi]\!]$ *can be done in time* $O(|\varphi| \cdot (|u| + |v|))$.

[4] As discussed in Remark 1 (Section 4), it is possible to extend the definition as well as our results to weighted atomic propositions with values in $[0,1]$.

2.1 LTL^{∇} Queries

Let $\mathcal{X}$ be a finite set of variables. An LTL^{∇} *query* (over $\mathcal{X}$) is an LTL^{∇} formula in which some of the quality constants are replaced with variables from $\mathcal{X}$. For example, $\varphi = \mathsf{G}(req \rightarrow ((\nabla_{x_1} X(read \vee \nabla_{x_2} X\, read)) \vee (\nabla_{x_3} X\, write) \vee (\nabla_{\frac{3}{4}} halt)))$ is an LTL^{∇} query over $\{x_1, x_2, x_3\}$. We say that an LTL^{∇} query is *simple* if each of its variables occurs only once. The *depth* of an LTL^{∇} query φ is the maximal nesting depth of variables in φ. For example, φ above is simple and is of depth 2. Note that, as in φ above, not all quality constraints are replaced by variables. For an LTL^{∇} query φ, we denote by $var(\varphi)$ the set of variables $x \in \mathcal{X}$ such that $\nabla_x \psi$ is a subformula of φ. Given an assignment $f : \mathcal{X} \rightarrow [0,1]$, we define φ^f to be the LTL^{∇} formula obtained from φ by replacing every occurrence of $x \in \mathcal{X}$ with $f(x)$. Note that an assignment f as above prioritizes the different possible ways to satisfy the specification. In φ above, the assignment to x_1 and x_3 reflects the priority of the designer as to whether a read lock or a write lock is granted after a request, and the assignment to x_2 reflects the cost of a delayed read lock.

Consider an LTL^{∇} query φ. A *path constraint* is a pair $\langle \pi, I \rangle$ such that $\pi \in (2^{AP})^{\omega}$ and $I \subseteq [0,1]$ is a closed interval; that is, $[a,b]$ for $0 \leq a \leq b \leq 1$. A *lasso constraint* is a path constraint in which π is a lasso. When the interval I is a single point, thus $a = b$, we only state the point in the specification of the constraint.

The LTL^{∇} query problem is to decide, given an LTL^{∇} query φ and a set $\mathcal{C}$ of lasso constraints, whether there exists an assignment f to $var(\varphi)$ such that $[\![\pi, \varphi^f]\!] \in I$ for all $\langle \pi, I \rangle \in \mathcal{C}$. We then say that the assignment f is a *solution* to $\langle \varphi, \mathcal{C} \rangle$.

2.2 Generating LTL^{∇} Queries

Our algorithms for solving the LTL^{∇} query problem takes as input an LTL^{∇} query, which is up to the designer to write. While the semantics of LTL^{∇} is easy to understand and use, there are some caveats one should be aware of when designing LTL^{∇} formulas and queries. In this section we demonstrate methods to soundly design LTL^{∇} queries. Moreover, the proposed methods can be automated, so that the designer may actually start with an LTL formula, rather than an LTL^{∇} query.

Typically, LTL^{∇} formulas are obtained from LTL formulas by adding the ∇_λ operator to various components. A common pattern for LTL specifications is a conjunction of properties. Consider a conjunction $\alpha \wedge \beta$. Assume that the satisfaction of β is less crucial than that of α. Specifically, if only β holds, we want the satisfaction value to be 0, but if only α holds, the satisfaction value is $\frac{3}{4}$. Note that a naive introduction of the ∇_λ operator may result in an undesirable behavior. In particular, according to the semantics of LTL^{∇}, the satisfaction value of $\alpha \wedge \nabla_{\frac{3}{4}} \beta$ is at most $\frac{3}{4}$, and the contribution of α, for values above $\frac{3}{4}$, is irrelevant. We now demonstrate two sound methods for adding ∇_λ operators.

As discussed in Section 1, different ways of satisfying a formula induce different qualities. A conjunction has only one way to be satisfied, thus in order to prioritize its components we decompose its components into a disjunction of cases, on which we apply the ∇_λ operator. For example, $\alpha \wedge \beta$ becomes $(\nabla_{\lambda_1}(\alpha \wedge \beta)) \vee (\nabla_{\lambda_2} \alpha) \vee$

$(\triangledown_{\lambda_3}\beta)$. After this transformation, the λ_i quality constants reflect the *gain* from the corresponding disjuncts.

The second method is to use negations to dualize the behavior of $\triangledown_\lambda$. Consider the formula $\neg\triangledown_{(1-\lambda)}\neg\varphi$ for some formula φ. Note that $[\![\pi, \neg\triangledown_{(1-\lambda)}\neg\varphi]\!] = 1-(1-\lambda)(1-[\![\pi,\varphi]\!])$. We use the abbreviation $\blacktriangledown_\lambda\varphi = \neg\triangledown_{(1-\lambda)}\neg\varphi$. In particular, if $[\![\pi,\varphi]\!] = 1$, then $\blacktriangledown_\lambda\varphi = 1$, and if $[\![\pi,\varphi]\!] = 0$, then $\blacktriangledown_\lambda\varphi = \lambda$. The operator $\blacktriangledown_\lambda$ does work well with conjunctions. For example, $\alpha \wedge \beta$ becomes $(\blacktriangledown_{\lambda_1}\alpha) \wedge (\blacktriangledown_{\lambda_2}\beta)$, with λ_i indicating the *loss* when the corresponding conjuncts do not hold. In the full version we formalize this intuition.

3 Solving the $\mathrm{LTL}^\triangledown$ Query Problem

In this section we study the complexity of the $\mathrm{LTL}^\triangledown$ query problem and show that it is NP-hard. As follows from Proposition 1, given an $\mathrm{LTL}^\triangledown$ query φ, a set $\mathcal{C}$ of lasso constraints, and an assignment $f : var(\varphi) \to [0,1]$, it is possible to check in linear time whether f is a solution for $\langle\varphi,\mathcal{C}\rangle$. Indeed, for each of the lasso constraints $\langle\pi, I\rangle \in \mathcal{C}$ we can calculate $[\![\pi,\varphi^f]\!]$ and verify that it is in I. This suggests that the $\mathrm{LTL}^\triangledown$ query is in NP, as given a witness assignment f, we can verify it efficiently. Membership in NP, however, also requires the witness f to be polynomial in the φ and $\mathcal{C}$.

The latter requirement adds to the picture considerations like the domain and representation of the interval constraints. A natural suggestion is to assume that all intervals I are of the form $[a,b]$ for rational numbers $0 \le a, b \le 1$, given by their binary expansion. As we now demonstrate, things are involved already in this case. To see why, consider the query $\triangledown_x\triangledown_x p$ and the constraint $\langle\{p\}^\omega, \frac{1}{2}\rangle$. The single solution to the problem is f with $f(x) = 1/\sqrt{2}$. But $1/\sqrt{2}$ is irrational, and therefore its binary expansion is infinite. Thus, while it is possible that the problem is in NP, describing a witness for an input requires a more sophisticated way of encoding solution, which is of debatable interest to the CAV community. As good news, in Section 4.1 we show that for typical instances of the problem, namely simple queries of depth 1, short witnesses exist, making the problem NP-complete for them. The proof requires results we develop in Section 4. Here, we describe the lower bound.

Theorem 1. *The* $\mathrm{LTL}^\triangledown$ *query problem is NP-hard.*

Proof: We describe a reduction from 3-SAT. Let $\theta = (l_1^1 \vee l_2^1 \vee l_3^1) \wedge ... \wedge (l_1^k \vee l_2^k \vee l_3^k)$ be a 3-CNF formula. We construct an $\mathrm{LTL}^\triangledown$ query φ and a set $\mathcal{C}$ of constraints such that θ is satisfiable iff there is a solution for $\langle\varphi,\mathcal{C}\rangle$. Let $X = \{x_1, ..., x_m\}$ be the set of variables that appear in θ. We define $AP = \{p_1, n_1, ..., p_m, n_m\}$ and $\mathcal{X} = \{y_1, z_1, \ldots, y_m, z_m\}$. Intuitively, the proposition p_i (resp. n_i) stands for "the variable x_i appears positively (resp. negatively) in the clause", and we define the query and the constraints so that the variable y_i (resp. z_i) is assigned 1 when x_i is assigned `True` (resp. `False`).

We define $\varphi = \mathsf{G}(\triangledown_{y_1}p_1 \vee \triangledown_{z_1}n_1 \vee ... \vee \triangledown_{y_m}p_m \vee \triangledown_{z_m}n_m)$. We first have to ensure that in every solution to the query, at least one of the variables $\{y_i, z_i\}$ gets value 0, for all $1 \le i \le m$. This is done by the constraint $\langle\pi_i, 0\rangle$, with $\pi_i = \{p_i\}\{n_i\}(AP)^\omega$. Note that in order for $[\![\pi_i,\varphi]\!]$ to be 0, it must be that either $[\![\{p_i\}, \triangledown_{y_i}p_i]\!] = 0$ or $[\![\{n_i\}, \triangledown_{z_i}n_i]\!] = 0$, implying that indeed at least one of the variables $\{y_i, z_i\}$ has value 0.

The family of m constraints above guarantees that a solution f to the query induces a truth assignment to X: the variable x_i is assigned `True` iff $f(y_i) = 1$. It is left to ensure that f induces a satisfying assignment. This is done by the constraint $\langle \pi, 1 \rangle$, where $\pi = \{s_1^1, s_2^1, s_3^1\} \cdots \{s_1^k, s_2^k, s_3^k\} \cdot (AP)^\omega$ is such the i-th position corresponds to the i-th clause and ensures that at least one of its literals gets value `True`. Accordingly, s_j^i is p_t if $l_j^i = x_t$ and is n_t if $l_j^i = \neg x_t$. Note that in order for $[\![\pi, \varphi]\!]$ to be 1, it must be that $[\![\{s_1^i, s_2^i, s_3^i\}, \triangledown_{y_t} p_t]\!] = 1$ or $[\![\{s_1^i, s_2^i, s_3^i\}, \triangledown_{z_t} z_t]\!] = 1$, for some t such that x_t appears in the i-th clause. If x_t appears positively in the clause, then one of the s_j^i's is p_t, and if x_t appears negatively, then one of them is n_t. Thus, in a solution f, one of the corresponding variables – that is, y_t in the first case and n_t in the second, is assigned 1.

It is easy to see that the reduction is polynomial.

□

Theorem 1 motivates a study of special easy cases of the LTL$^\triangledown$ query problem. Since the reduction in the proof of Theorem 1 uses a query with multiple constraints, a natural candidate is the case of a single constraint. Lemma 1 below hints that this case is not easier.

Lemma 1. *Let φ be a simple* LTL$^\triangledown$ *query over a set $\mathcal{X}$ of variables and let $\mathcal{C}$ be a set of lasso constraints of the form $\langle \pi, 1 \rangle$ or $\langle \pi, 0 \rangle$. Then, there exists an* LTL$^\triangledown$ *query φ' over $\mathcal{X}$ and a lasso π such that for every assignment $f : \mathcal{X} \to [0,1]$, we have that f is a solution to $(\varphi', \{\langle \pi, 0 \rangle\})$ iff f is a solution to $\langle \varphi, \mathcal{C} \rangle$. In addition, the length of the prefix of π is the length of the longest prefix of a lasso in $\mathcal{C}$, and the length of its cycle is the* lcm *(least common multiple) of the lengths of the cycles in the lassos in $\mathcal{C}$.*

Proof: Let AP be the set of atomic propositions in φ, and let $\mathcal{C} = \{\langle u_1 \cdot v_1^\omega, c_1 \rangle, \ldots, \langle u_k \cdot v_k^\omega, c_k \rangle\}$. We define AP' as k disjoint copies of AP, thus $AP' = AP \times \{1, \ldots, k\}$. We define φ_j to be the LTL$^\triangledown$ query obtained from φ by replacing each atomic proposition $p \in AP$ by the atomic proposition $\langle p, j \rangle \in AP'$. Let $u, v \in 2^{AP'}$ be such that for all $1 \leq j \leq k$, the projection of $u \cdot v^\omega$ on $AP \times \{j\}$ agrees with $u_j \cdot v_j^\omega$. It is easy to define u and v as above by taking u of length $m = \max_i |u_i|$ and v of length $\ell = lcm(|v_1|, ..., |v_k|)$. Indeed, the labeling of u by $AP \times \{j\}$ is obtained by concatenating to u_j a prefix of v_j^ω of length $m - |u_j|$ and the labeling of v by $AP \times \{j\}$ is then obtained from v_j' by the corresponding shift of $v_j^{\ell/|v_j|}$.

Now, we define $\varphi' = \bigwedge_{j=1}^k \psi_j$, where ψ_j is either φ_j, in case $c_j = 1$, or is $\neg\varphi_j$, in case $c_j = 0$. Consider an assignment $f : \mathcal{X} \to [0,1]$. Since φ' is defined as a conjunction, then the constraint $\langle u \cdot v^\omega, 1 \rangle$ is met for φ' iff $[\![u \cdot v^\omega, \psi_j^f]\!] = 1$ for all $1 \leq j \leq k$, which, by the definition of ψ_j, holds iff f is a solution to $\langle \varphi, \{\langle u_j \cdot v_j^\omega, c_j \rangle\} \rangle$. To conclude, f is a solution to $\langle \varphi', \{\langle u \cdot v^\omega, 1 \rangle\} \rangle$ iff f is a solution to $\langle \varphi, \mathcal{C} \rangle$. □

While the single lasso constructed in Lemma 1 may be exponential in the original constraints, examining the lassos that are used in the reduction in the proof of Theorem 1, we see that they all have cycles of length 1. Therefore, Lemma 1 together with the reduction there imply that the special case of a single constraint is not easy. Formally, we have the following.

Theorem 2. *The* $\mathrm{LTL}^{\triangledown}$ *query problem is NP-hard even for the case of a single lasso constraint.*

4 A Feasible Special Case

While Theorem 2 implies that the $\mathrm{LTL}^{\triangledown}$ query problem is hard already for a single constraint, the transformation described in the proof of Lemma 1 generates formulas that are not simple. Indeed, the transformation is based on a relation between multiple constraints and multiple occurrences of a variable. In this section we show that in a setting with both limitations, the $\mathrm{LTL}^{\triangledown}$ query problem can be solved efficiently. Formally, we prove the following.

Theorem 3. *The* $\mathrm{LTL}^{\triangledown}$ *query problem for simple queries and a single constraint can be solved in polynomial time.*

In Section 5, we show that Theorem 3 and the algorithm developed for its proof are useful in approximation and heuristic algorithms for the general case.

Let φ be a simple $\mathrm{LTL}^{\triangledown}$ query over AP and $\mathcal{X}$, and let π be a computation. Let $k = |\mathcal{X}|$. Consider the function $\mu_{\pi,\varphi} : [0,1]^k \to [0,1]$ defined by $\mu_{\pi,\varphi}(f) = [\![\pi, \varphi^f]\!]$.

We start with some useful observations.

Lemma 2. *For all computations* π *and* $\mathrm{LTL}^{\triangledown}$ *queries* φ*, the function* $\mu_{\pi,\varphi}$ *is continuous. That is, for every infinite sequence* $(a_n)_{n=1}^{\infty}$ *of points in* $[0,1]^k$ *such that* $\lim_{n\to\infty} a_n = a$*, it holds that* $\lim_{n\to\infty}(\mu_{\pi,\varphi}(a_n)) = \mu_{\pi,\varphi}(a)$.

Consider a variable $x \in \mathcal{X}$. Since φ is simple, the variable x is either *positive* in φ, in case the subformula $\triangledown_x \psi$ is in the scope of an even number of negation, or is *negative* in φ, otherwise. We refer to the positivity or negativity of x in φ as its *polarity* in φ.

Lemma 3. *For all computations* π *and* $\mathrm{LTL}^{\triangledown}$ *queries* φ*, the function* $\mu_{\pi,\varphi}$ *is monotonic in each variable. Specifically, for every variable* $x \in \mathcal{X}$*, if* x *is positive in* φ *then* $\mu_{\pi,\varphi}$ *is increasing with* x *and if* x *is negative in* φ *then* $\mu_{\pi,\varphi}$ *is decreasing with* x*.*

We note that proofs of Lemmas 2 and 3 are by induction on the structure of φ.

The idea behind our polynomial algorithm is to limit the search space for a satisfying assignment. Before defining the limited search space, let us first observe that an $\mathrm{LTL}^{\triangledown}$ formula has finitely (in fact, linearly many) possible satisfaction values. We define the set of possible values of φ, denoted $val(\varphi)$, by induction on the structure of φ as follows.

- If $\varphi = p \in AP$, then $val(p) = \{0, 1\}$.
- If $\varphi = \psi_1 \vee \psi_2$ or $\varphi = \psi_1 U \psi_2$, then $val(\varphi) = val(\psi_1) \cup val(\psi_2)$.
- If $\varphi = \neg\psi$, then $val(\varphi) = \{1 - v : v \in val(\psi)\}$.
- If $\varphi = \mathsf{X}\psi$, then $val(\varphi) = val(\psi)$.
- If $\varphi = \triangledown_\lambda \psi$, then $val(\varphi) = \{\lambda \cdot v : v \in val(\psi)\}$.

It is easy to prove that for every path π it holds that $[\![\pi, \varphi]\!] \in val(\varphi)$.

We start by defining the limited search space for $\mathrm{LTL}^{\triangledown}$ queries of depth 1. We will later generalize the definition to all depths. Let φ be a simple $\mathrm{LTL}^{\triangledown}$ query of depth 1. For $x \in var(\varphi)$ and $c \in [0,1]$ we define the set of relevant values for x with respect to φ and c, denoted $val(x, \varphi, c)$, by induction on the structure of φ as follows.

- If $\varphi = \nabla_x \psi$, for a LTL$^\nabla$ formula ψ, then $val(x, \varphi, c) = \{\frac{c}{v} : v \in val(\psi) \text{ and } v \geq c\}$. Note that since φ is of nesting depth 1, then ψ has no variables, and this is the base case for the induction.
- If $\varphi = \psi_1 \vee \psi_2$ or $\varphi = \psi_1 U \psi_2$, then $val(x, \varphi, c) = val(x, \psi_i, c)$, for the single $i \in \{1, 2\}$ such that $x \in var(\psi_i)$.
- If $\varphi = \mathsf{X}\psi$, then $val(x, \varphi, c) = val(x, \psi, c)$.
- If $\varphi = \neg\psi$, then $val(x, \varphi, c) = val(x, \psi, 1 - c)$.
- If $\varphi = \nabla_\lambda \psi$, we distinguish between two cases. If $\lambda \geq c$, then $val(x, \varphi, c) = val(x, \psi, \frac{c}{\lambda})$. Otherwise, $val(x, \varphi, c) = \emptyset$.

Lemma 4 below justifies the restricted search space. Consider a value $u \in [0, 1]$. Let $up_{x,\varphi,c}(u)$ and $down_{x,\varphi,c}(u)$ be the "rounding" up and down of u to the nearest value in $val(x, \varphi, c)$. Formally, $up_{x,\varphi,c}(u) = \min\{v : v \in val(x, \varphi, c) \text{ and } v \geq u\}$ and $down_{x,\varphi,c}(u) = \max\{v : v \in val(x, \varphi, c) \text{ and } v \leq u\}$.

Consider an assignment $f : \mathcal{X} \to [0, 1]$. For a variable $x \in \mathcal{X}$, define the assignments $f^+_{x,\varphi,c}$ and $f^-_{x,\varphi,c}$ as the assignments obtained from f by leaving the assignments to all variables except x unchanged and rounding the value of x up or down to the nearest value in $val(x, \varphi, c)$. The decision whether to round the value of x up or down depends on the $+$ and $-$ indication as well as in the polarity of x in φ. Formally, we have the following.

$$f^+_{x,\varphi,c}(x) = \begin{cases} up_{x,\varphi,c}(f(x)) & \text{if } x \text{ is positive in } \varphi, \\ down_{x,\varphi,c}(f(x)) & \text{if } x \text{ is negative in } \varphi, \end{cases}$$

and dually (switch positive and negative) for $f^-_{x,\varphi,c}(x)$.

Lemma 4. *Consider a simple* LTL$^\nabla$ *query* φ *of depth* 1 *and an assignment* f *to* $var(\varphi)$. *Let* $c \in [0, 1]$, *and let* π *be a computation. Then,*

1. *If* $[\![\pi, \varphi^f]\!] \leq c$, *then* $[\![\pi, \varphi^{f^+_{x,\varphi,c}}]\!] \leq c$.
2. *If* $[\![\pi, \varphi^f]\!] \geq c$, *then* $[\![\pi, \varphi^{f^-_{x,\varphi,c}}]\!] \geq c$.

Note that by the monotonicity of LTL$^\nabla$ queries, increasing the value of a variable x that appears positively in φ can only increase the satisfaction value of φ (and dually for reducing the value of x or for the case of a variable that appears negatively). The claim in Lemma 4, however, is different and is much stronger, as it states that we can actually increase the value of a variable that appears positively without increasing the value of φ. More precisely, if the satisfaction value of φ^f in π is below c, then we can round the value of x up to the closest value in $val(x, \varphi, c)$ and still keep the satisfaction value below c.

We can now prove that the restriction of the search space to values in $val(x, \varphi, c)$ is allowed.

Lemma 5. *Let* φ *be a simple* LTL$^\nabla$ *query of depth* 1, *let* $c \in [0, 1]$, *and let* π *be a path. If there exists an assignment* f *such that* $[\![\pi, \varphi^f]\!] = c$, *then there also exists an assignment* g *such that for every* $x \in var(\varphi)$ *it holds that* $g(x) \in val(x, \varphi, c)$ *and* $[\![\pi, \varphi^g]\!] = c$.

Proof: Consider a simple LTL$^{\triangledown}$ query φ of depth 1 and an assignment f to $var(\varphi)$. Let $c \in [0,1]$, and let π be a computation. By the monotonicity of $\mu_{\pi,\varphi}$, Lemma 4 implies that if $[\![\pi, \varphi^f]\!] = c$, then $[\![\pi, \varphi^{f^+_{x,\varphi,c}}]\!] = c$.

Let f be such that $[\![\pi, \varphi^f]\!] = c$. The assignment g is obtained by repeating the following process for all variables $x \in var(\varphi)$ in an arbitrary order: if $f(x) \in val(x, \varphi, c)$, then $g(x) = f(x)$. Otherwise, we define $g(x)$ to be $f^+_{x,\varphi,c}(x)$. By the above, $[\![\pi, \varphi^g]\!] = c$. □

Consider a simple LTL$^{\triangledown}$ query φ over $\mathcal{X}$. By Lemma 5, the search for a solution f to a single constraint with a point interval c involves a search in finitely many possible assignments – these that map each variable $x \in var(\varphi)$ to values in $val(x, \varphi, c)$. By Lemma 3 (monotonicity of assignments), the search can combine a binary search for the assignment for each $x \in var(\varphi)$ with an ordering of the different variables. We will get back to this point when we describe our experimental results in Section 6.

The search described above assumes simple queries of depth 1 and constraints with point intervals. We now remove both assumptions. Let f^{max} and f^{min} be the assignments that maximizes and minimizes the value of φ. Thus, $f^{max}(x)$ is 1 if x appears positively in φ and is 0 otherwise, and dually for f^{min}. We define the LTL$^{\triangledown}$ formulas $\varphi^{max} = \varphi^{f^{max}}$ and $\varphi^{min} = \varphi^{f^{min}}$. Note that $\triangledown_1\psi$ and $\triangledown_0\psi$ subformulas can be replaced by ψ and `False`, respectively.

For a simple LTL$^{\triangledown}$ query φ (of an arbitrary depth), let $\varphi^\star$ be the LTL$^{\triangledown}$ formula obtained from φ by replacing every subformula of the form $\triangledown_x\psi$ by $\triangledown_x\psi^{max}$. Note that $\varphi^\star$ is of depth 1. By Lemma 3 (monotonicity of assignments), for every computation π, LTL$^{\triangledown}$ query ψ, and assignment f, we have $[\![\pi, \psi^f]\!] \leq [\![\pi, \psi^{max}]\!]$. Thus, replacing ψ by ψ^{max} may cause the satisfaction value of ψ to go above a desired bound. As we show below, however, in this case we can play with the assignment to x in order "tune down" the satisfaction value of ψ^{max}.

Lemma 6. *Let φ be a simple* LTL$^{\triangledown}$ *query, and let $\langle \pi, I\rangle$ be a constraint. The* LTL$^{\triangledown}$ *query $\langle \varphi, \langle \pi, I\rangle\rangle$ has a solution iff the query $\langle \varphi^\star, \langle \pi, I\rangle\rangle$ has a solution.*

Lemma 6 implies that we can use our algorithm also for formulas of depth greater than 1 by applying the algorithm to $\varphi^\star$. It is left to extend the algorithm to handle interval constraints. That is, given a simple LTL$^{\triangledown}$ query φ and a lasso constraint $\langle \pi, I\rangle$ such that $I \subseteq [0,1]$, our goal is to decide whether $\langle \varphi, \langle \pi, I\rangle\rangle$ has a solution. We do this as follows. First, compute $a = [\![\pi, \varphi^{min}]\!]$ and $b = [\![\pi, \varphi^{max}]\!]$. If $I \cap [a,b] = \emptyset$, then, as $[\![\pi, \psi^{min}]\!] \leq [\![\pi, \psi^f]\!] \leq [\![\pi, \psi^{max}]\!]$, we can conclude that there is no solution to $\langle \varphi, \langle \pi, I\rangle\rangle$. Otherwise, by the continuity of $\mu_{\pi,\varphi}$, every value in $c \in [a,b]$ can be attained by $\mu_{\pi,\varphi}$, so one can choose any value $c \in I \cap [a,b]$, and find a solution to $\langle \varphi, \langle \pi, c\rangle\rangle$.

Remark 1. Our definition of computations assumes Boolean atomic propositions. It is easy to extend our setting and results to computations over *weighted atomic propositions*. Let WP be a finite set of weighted atomic propositions over some domain D. The domain D may be infinite (say, the natural numbers) and different propositions may be over different domains. Each position in the computation is then a function in D^{WP}, and the semantics of LTL$^{\triangledown}$ is as in the Boolean case, except that $[\![\pi, p]\!]$, for $p \in WP$

is $\pi_0(p)$. The solution of the corresponding LTL$^\triangledown$ query is similar to the one described above, except that the definition of $val(\psi)$, and consequently also $val(x, \varphi, c)$, should be adjusted to reflect the fact the weighted propositions in ψ can take values in D. Since each lasso commutation has only finitely many positions, the number of the values to be considered is still linear in the input to the problem.

4.1 NP Completeness of the LTL$^\triangledown$ Query Problem for Simple Queries

Lemma 5 gives us an upper bound to complete Theorem 1 for simple LTL$^\triangledown$ queries of depth 1. For an LTL$^\triangledown$ query φ, a set of constraints $\mathcal{C} = \{\langle \pi_i, [a_i, b_i]\rangle\}_{i=1}^m$, and a variable x, define $val(x, \varphi, \mathcal{C}) = \bigcup_{i=1}^m val(x, \varphi, b_i)$. That is, $val(x, \varphi, \mathcal{C})$ includes the restricted search space of the upper end points of the intervals in the constraints in $\mathcal{C}$.

Lemma 7. *Let φ be a simple* LTL$^\triangledown$ *query of depth* 1*, and let $\mathcal{C}$ be a set of constraints. If there exists a solution f for $\langle \varphi, \mathcal{C}\rangle$, then there also exists a solution g such that for every $x \in var(\varphi)$, it holds that $g(x) \in val(x, \varphi, \mathcal{C})$.*

Since the binary expansion of the values in $val(x, \varphi, \mathcal{C})$ is polynomial in φ and $\mathcal{C}$ (with intervals given by their binary expansion), Lemma 7 implies that a witness solution to a simple LTL$^\triangledown$ query of depth 1 is polynomial. Since witnesses can be verified in linear time, we can conclude with the following.

Theorem 4. *The* LTL$^\triangledown$ *query problem for simple queries of depth* 1 *is NP-complete.*

5 Approximations and Heuristics

In this section we discuss two heuristic schemes for the LTL$^\triangledown$ query problem. The motivation is twofold. First, our heuristic algorithms run in polynomial time, whereas, as studied in Section 3, the problem is NP-hard. Second, in case a query does not have a solution, it is helpful to find a sub-optimal, or partial, solution. In order to study sub-optimal solutions, we should first formalize the LTL$^\triangledown$ query problem as an optimization problem. To do so, it is convenient to consider the problem in a geometrical perspective: for an LTL$^\triangledown$ query φ and a set of constraints $\mathcal{C}$, let $k = |var(\varphi)|$. Every constraint $\langle \pi, I\rangle$ induces a set $S_{\langle \pi, I\rangle, \varphi} \subseteq [0, 1]^k$ of solutions to $\langle \pi, I\rangle$. The LTL$^\triangledown$ query problem then amounts to finding a point $f \in \bigcap_{\langle \pi, I\rangle \in \mathcal{C}} S_{\langle \pi, I\rangle, \varphi}$. We note that the main difficulty in solving the LTL$^\triangledown$ query problem is that even for a single constraint, this set may not be convex, and therefore not amenable to methods of convex analysis. Indeed, consider for example the query $\psi = (\triangledown_x p) \vee (\triangledown_y q)$ with the constraint $\langle \{p, q\}^\omega, 1\rangle$. We have that $S_{\mathcal{C}, \psi} = ([0, 1] \times \{1\}) \cup (\{1\} \times [0, 1])$, which is not convex.

Using the geometrical perspective, we consider the following optimization problem: for an LTL$^\triangledown$ query φ and a set of constraints $\mathcal{C}$, find an assignment that minimizes the sum (over $\langle \pi, I\rangle \in \mathcal{C}$) of distances to $S_{\langle \pi, I\rangle, \varphi}$. We still have some freedom in choosing a distance function. Since we evaluate an assignment by the satisfaction value it induces on a computation, it is natural to use the obtained satisfaction value as an underlying metric for the distance function. Traditionally, such distance functions are known as *loss functions*, and we consider two common ones here. Let $x, y \in [0, 1]$.

- $0/1$-*loss*, defined by $\ell_{0/1}(x,y) = 0$ if $x = y$, and $\ell_{0/1}(x,y) = 1$ if $x \neq y$.
- $\|\cdot\|_2$-*loss*, defined by $\|x,y\|_2 = |x-y|$.

Consider a loss function ℓ and an assignment $f \in [0,1]^k$. Recall that for a closed interval $A \subseteq [0,1]$ and $x \in [0,1]$, we have $\ell(x,A) = \min\{\ell(x,y) : y \in A\}$. We define $dist_\ell(\varphi, \mathcal{C}, f) = \frac{1}{|\mathcal{C}|}\sum_{\langle\pi,I\rangle\in\mathcal{C}} \ell([\![\pi, \varphi^f]\!], I)$. That is, the average loss. Then, the $\mathrm{LTL}^\triangledown$ query optimization problem is to find $\arg\min_{f\in[0,1]^k}\{dist(\varphi,\mathcal{C},f)\}$, namely an assignment that minimizes the loss. Accordingly, for $\ell_{0/1}$, the problem is to find an assignment that maximizes the number of satisfied constraints, and in $\|\cdot\|_2$-loss we want to minimize the geometrical distance.

Note that for both loss functions ℓ, an assignment f is a solution iff $dist_\ell(\varphi,\mathcal{C},f) = 0$. Hence, the problem of finding the optimum is NP-hard. As Theorem 2 shows, the $\mathrm{LTL}^\triangledown$ query problem is NP-hard even when a single constraint is allowed. Accordingly, since a nontrivial approximation of the number of satisfied constraints must be greater than 0, it is NP-hard to even approximate the problem (to any ratio) under $\ell_{0/1}$.

Our use of $\mathrm{LTL}^\triangledown$ queries for generating quality specifications makes $\ell_{0/1}$ less appealing. Indeed, it takes us back to the Boolean setting. Still it involves some nice theoretical aspects. In particular, as we show in the full version, it suggests a naive $\frac{1}{2}$-approximation (that is, a guarantee that at most hard of the constraints are satisfied) for the case the constraints are all pure upper- or lower-bounds.

As we mentioned above, the set of solutions (even when it is not empty) may not be convex, which is the underlying reason for the hardness of the problem. We suggest a polynomial-time heuristic algorithm, based on our ability to solve the $\mathrm{LTL}^\triangledown$ query problem efficiently for a single constraint (Theorem 3). Given a simple $\mathrm{LTL}^\triangledown$ query φ and a set $\mathcal{C}$ of constraints, we find, for every constraint $\langle\pi, I\rangle \in \mathcal{C}$, an assignment $f_{\langle\pi,I\rangle}$ such that $[\![\pi, \varphi^{f_{\langle\pi,I\rangle}}]\!] \in I$. Let $F = \{f_{\langle\pi,I\rangle} : \langle\pi,I\rangle \in \mathcal{C}\}$. Intuitively, every point $f \in F$ "represents" the set $S_{\langle\pi,I\rangle,\varphi}$ for one of the constraints. Our algorithm combines the points in F in order to obtain a single assignment. If the representation is "good enough", we hope to get a good assignment. There are several ways to obtain a single assignment from F. Our algorithm uses the following three.

- *Minimum assignment*, denoted f_{min}: For every $x \in var(\varphi)$, if x is positive in φ, then $f_{min}(x) = \min_{f\in F} f(x)$; otherwise $f_{min}(x) = \max_{f\in F} f(x)$.
- *Maximum assignment*, denoted f_{max}: For every $x \in var(\varphi)$, if x is positive in φ, then $f_{max}(x) = \max_{f\in F} f(x)$; otherwise $f_{max}(x) = \min_{f\in F} f(x)$.
- *Center of gravity*, denoted f_{CoG}: For every $x \in var(\varphi)$, we define $f_{CoG}(x) = \frac{1}{|F|}\sum_{f\in F} f(x)$. It is easy to prove that the center of gravity is the point that minimizes the square-distance from $f_1, ..., f_m$. That is, $f_{CoG} = \arg\min\{\sum_{i=1}^m \|f - f_i\|_2^2 : f \in [0,1]^k\}$, where $\|x\|_2 = \sqrt{\sum_{i=1}^k x_i^2}$ for $x \in [0,1]^k$. This motivates using this as a heuristic, as it minimizes some sort of distance to the constraints.

Before we proceed to the experimental results, we show that in theory, all three methods may perform poorly.

Example 1. Recall the example $\psi = (\triangledown_x p) \vee (\triangledown_y q)$ with the set of constraints $\mathcal{C} = \{\langle\{p\}^\omega, 1\rangle, \langle\{q\}^\omega, 1\rangle\}$. Clearly, the assignment $(1,1)$ is a solution. Assume that our

polynomial time algorithm returns $F = \{(1,0),(0,1)\}$; that is, each point is a solution to a single constraint. The minimum assignment heuristic then gives $f_{min} = (0,0)$, for which $dist_{\ell_{0/1}}(\psi, \mathcal{C}, f_{min}) = dist_{\|\cdot\|_2}(\psi, \mathcal{C}, f_{min}) = 1$. The CoG heuristic cannot do quite as badly, being an average. However, in the example above we have $f_{CoG} = (\frac{1}{2}, \frac{1}{2})$ for which $dist_{\ell_{0/1}}(\psi, \mathcal{C}, f_{CoG}) = 1$ and $dist_{\|\cdot\|_2}(\psi, \mathcal{C}, f_{CoG}) = \frac{1}{2}$. It is easy to verify that taking the query $\neg\psi$ with the same constraints may result again in $F = \{(1,0),(0,1)\}$, whereas the optimum now is $(0,0)$, and $f_{max} = (1,1)$, again giving a bad lower bound.

6 Experimental Results

In this section we present experimental results demonstrating our heuristic algorithm for solving the LTL$^\nabla$ query problem. We evaluate the quality of the heuristic under the two loss functions (namely $\ell_{0/1}$ and $\|\cdot\|_2$). As our benchmark we use LTL formulas from [12], to which we add a quality layer, as well as queries constructed manually. As constraints, we use accepting paths in the automaton for the corresponding LTL formulas, as well as manually generated computations. An example of a query is a specification for a traffic light. The atomic propositions are $\{N, E, W, S\}$, standing for a green light for traffic coming from North, East, West, and South, respectively. We assume traffic crosses the junction and makes no turns, and so the specification allows N and S, as well as W and E, to hold simultaneously, but not N and W, nor S and E, and so on. Thus, the specification includes the property $\psi = (G(N \vee S) \rightarrow (\neg E \wedge \neg W)) \wedge (G(E \vee W) \rightarrow (\neg N \wedge \neg S))$. We want the traffic light to direct the traffic efficiently, so we require that at least one direction has a green light. This is specified by the conjunct $G\theta$, for $\theta = (S \vee N \vee W \vee E)$. We may also be satisfied by $FG\theta$. But while ψ is a crucial safety requirement, the conjuncts involving θ only concern the efficiency of the traffic light. Thus, we can tune them down using the LTL$^\nabla$ formula $\psi \wedge \blacktriangledown_{0.4} G\theta \wedge \blacktriangledown_{0.9} FG\theta$ (as discussed in Section 2.2, we use the abbreviation $\blacktriangledown_\lambda \varphi = \neg\nabla_{(1-\lambda)}\neg\varphi$ to indicating the *loss* when the corresponding conjuncts do not hold). Note that we prefer a junction that is never empty over a junction that is only eventually never empty. But this is not the end of the story. We may want to prioritize the different directions. Deciding the priorities and their combination with the external tuning down of the "efficiency requirement" may be a difficult task. So, we replace θ by $\theta' = (\blacktriangledown_{x_1} S) \vee (\blacktriangledown_{x_2} N) \vee (\blacktriangledown_{x_3} W) \vee (\blacktriangledown_{x_3} E)$, leaving the priorities as variables. The obtained LTL$^\nabla$ query is φ_5 in the table. Examples of constraints we use are $\langle(\{N,S\},\{E,W\})^\omega, 1\rangle$ and $\langle(\{N,S\},\emptyset,\{E,W\})^\omega, 0.4\rangle$.

We implemented the algorithm in Python and ran it on an Intel® Core i5 2.53GHz machine. The code can be found in: http://www.cs.huji.ac.il/∼guya03/CAV13/.

In Table 2 we evaluate the quality of results in the $\ell_{0/1}$ loss function. The results show that the algorithm preforms very well compared to the optimum. We ran the heuristic algorithm 40 times, each time with a different random ordering of the variables (recall that the algorithm for the case of a single constraint chooses variables according to an arbitrary order, which affects the resulting assignment). In each run we calculate, f_{min} and f_{max} in each iteration. We evaluate the assignments by checking how many constraints they satisfy, and output the best assignment. The running time of the algorithm that is shown in the table is the total running time, which is still negligible compared

to the optimal algorithm's running time.[5] In order to find the optimal assignment we go over all the assignments that use values in the restricted search space, hence the very high running times.

Table 2. Evaluating the heuristic under the $\ell_{0/1}$ loss

Query	$\|\mathcal{C}\|$	$\|\mathcal{X}\|$	Algorithm		Optimum	
			# of constraints satisfied	running time	# of constraints satisfied	Running time
φ_1	8	4	5	10sec	6	35sec
φ_2	6	4	3	27sec	3	151sec
φ_3	8	4	1	37sec	2	202sec
φ_4	4	5	2	11sec	2	260sec
φ_5	11	8	2	21sec	6	620sec

We continue to evaluate the heuristic under the $\|\cdot\|_2$ loss function, as shown in Table 3. In this approach we have no optimum to compare with (as we do not even know that the problem is in NP). Instead, we perform *sanity checks* and compare our results to them. As in the previous table, we run the algorithm 40 times and calculate f_{min}, f_{max}, and f_{CoG}. We evaluate the assignments by calculating $dist_{\|\cdot\|_2}(\varphi, \mathcal{C}, f_{min})$, $dist_{\|\cdot\|_2}(\varphi, \mathcal{C}, f_{max})$, and $dist_{\|\cdot\|_2}(\varphi, \mathcal{C}, f_{CoG})$, which we present in the table. Our first sanity check is the *Cross validation* technique, which is a widely used technique in machine learning. We partition the constraints $\mathcal{C}$ into two sets: $\mathcal{C}_1$ and $\mathcal{C}_2$. We find an assignment f_1 using the constraints $\mathcal{C}_1$. Then, we evaluate the assignments on the constraints $\mathcal{C}_2$. That is, we calculate $dist_{\|\cdot\|_2}(\varphi, \mathcal{C}_2, f_1)$. In machine learning, the goal of this technique is, given a training set, to assess the quality of a hypothesis. The difference between the learning scenario and our case is that there, the training set is chosen uniformly from a certain distribution, whereas our constraints are manually chosen by the designer. Thus, the accuracy of this assessment is strongly affected by the dependencies between the constraints. This makes cross-validation unreliable at times.

Table 3. Evaluating the heuristic in the distance-minimization approach

Query	$\|\mathcal{C}\|$	$\|\mathcal{X}\|$	Distance			Sanity checks			
			Min	Max	CoG	Min	Max	CoG	random
φ_1	8	4	0.031	0.031	0.129	0.05	0.2	0.218	0.230
φ_2	6	4	0.333	0.166	0.219	0.466	0.466	0.466	0.322
φ_3	8	4	0.229	0.104	0.162	0.191	0.275	0.4	0.200
φ_4	4	5	0.05	0.05	0.075	0.175	0.075	0.225	0.096
φ_5	11	8	0.1	0.173	0.1	0.34	0.06	0.127	0.348

In our second sanity check, we calculate $dist_{\|\cdot\|_2}(\varphi, \mathcal{C}, f)$ for a random assignment f. We repeat this test 10 times and take the average distance, thus approximating the expectancy of the distance.

[5] In practice, the lassos in the constraints are typically short, as the designer grades them manually according to specific behaviors he has in mind. We still challenged our implementation with both short and long lassos, and running time was not an issue – what we care here more is the quality of the assignment returned.

As seen in the table, there is no clear winner between the minimal, maximal, and center of gravity mechanism. As the running time of the algorithm is very short, there is no reason not to find all three points and choose the best one for the specific instance of the problem. As described above, the first sanity check returns mixed results. However, our heuristic significantly out-performs the random assignment.

References

1. Almagor, S., Boker, U., Kupferman, O.: Discounting in LTL. TR (2013), http://leibniz.cs.huji.ac.il/tr/1300.pdf
2. Almagor, S., Boker, U., Kupferman, O.: Formalizing and reasoning about quality. ICALP 2013 (2013)
3. Alur, R., Henzinger, T.A., Vardi, M.Y.: Parametric real-time reasoning. In: Proc. 25th STOC, pp. 592–601 (1993)
4. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
5. Ammons, G., Mandelin, D., Bodk, R., Larus, J.R.: Debugging temporal specifications with concept analysis. In: Proc. PLDI, pp. 182–195. Springer (2003)
6. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
7. Avni, G., Kupferman, O.: Parameterized Weighted Containment. In: Pfenning, F. (ed.) FOSSACS 2013. LNCS, vol. 7794, pp. 369–384. Springer, Heidelberg (2013)
8. Bloem, R., Cavada, R., Pill, I., Roveri, M., Tchaltsev, A.: RAT: A tool for the formal analysis of requirements. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 263–267. Springer, Heidelberg (2007)
9. Boker, U., Chatterjee, K., Henzinger, T.A., Kupferman, O.: Temporal specifications with accumulative values. In: Proc. 26th LICS, pp. 43–52 (2011)
10. Chan, W.: Temporal-logic queries. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 450–463. Springer, Heidelberg (2000)
11. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: ICSE, pp. 411–420 (1999)
12. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: Proc. FMCAD, pp. 117–124 (2006)
13. Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)
14. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
15. Kupferman, O.: Sanity checks in formal verification. In: Baier, C., Hermanns, H. (eds.) CONCUR 2006. LNCS, vol. 4137, pp. 37–51. Springer, Heidelberg (2006)
16. Kwiatkowska, M.Z.: Quantitative verification: models techniques and tools. In: ESEC/SIGSOFT FSE, pp. 449–458 (2007)
17. Madhusudan, P.: Learning algorithms and formal verification (Invited tutorial). In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 214–214. Springer, Heidelberg (2007)
18. Moon, S., Lee, K., Lee, D.: Fuzzy branching temporal logic. IEEE Transactions on Systems, Man, and Cybernetics, Part B 34(2), 1045–1055 (2004)
19. Pnueli, A.: The temporal semantics of concurrent programs. TCS 13, 45–60 (1981)
20. PROSYD. The Prosyd project on property-based system design, http://www.prosyd.org
21. Roveri, M.: Novel techniques for property assurance. TR PROSYD FP6-IST-507219 (2007)
22. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: Proc. PLDI, pp. 281–294 (2005)

Upper Bounds for Newton's Method on Monotone Polynomial Systems, and P-Time Model Checking of Probabilistic One-Counter Automata*

Alistair Stewart[1], Kousha Etessami[1], and Mihalis Yannakakis[2]

[1] School of Informatics, University of Edinburgh
kousha@inf.ed.ac.uk, stewart.al@gmail.com
[2] Department of Computer Science, Columbia University
mihalis@cs.columbia.edu

Abstract. A central computational problem for analyzing and model checking various classes of infinite-state recursive probabilistic systems (including quasi-birth-death processes, multi-type branching processes, stochastic context-free grammars, probabilistic pushdown automata and recursive Markov chains) is the computation of *termination probabilities*, and computing these probabilities in turn boils down to computing the *least fixed point* (LFP) solution of a corresponding *monotone polynomial system* (MPS) of equations, denoted $x = P(x)$.

It was shown by Etessami and Yannakakis [11] that a decomposed variant of Newton's method converges monotonically to the LFP solution for any MPS that has a non-negative solution. Subsequently, Esparza, Kiefer, and Luttenberger [7] obtained upper bounds on the convergence rate of Newton's method for certain classes of MPSs. More recently, better upper bounds have been obtained for special classes of MPSs ([10, 9]).

However, prior to this paper, for arbitrary (not necessarily strongly-connected) MPSs, no upper bounds at all were known on the convergence rate of Newton's method as a function of the encoding size $|P|$ of the input MPS, $x = P(x)$.

In this paper we provide worst-case upper bounds, as a function of both the input encoding size $|P|$, and $\epsilon > 0$, on the number of iterations required for decomposed Newton's method (*even with rounding*) to converge to within additive error $\epsilon > 0$ of q^*, for an *arbitrary* MPS with LFP solution q^*. Our upper bounds are essentially optimal in terms of several important parameters of the problem.

Using our upper bounds, and building on prior work, we obtain the first P-time algorithm (in the standard Turing model of computation) for quantitative model checking, to within arbitrary desired precision, of discrete-time QBDs and (equivalently) probabilistic 1-counter automata, with respect to any (fixed) ω-regular or LTL property.

* A full version of this paper is available at arxiv.org/abs/1302.3741. Research partially supported by NSF Grant CCF-1017955.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 495–510, 2013.

1 Introduction

In recent years, there has been extensive work on the analysis of various classes of infinite-state recursive probabilistic systems, including recursive Markov chains, probabilistic pushdown systems, stochastic context-free grammars, multi-type branching processes, quasi-birth-death processes and probabilistic 1-counter automata (e.g. [11, 12, 8, 9, 10, 4]). These are all finitely-presentable models that specify an infinite-state underlying probabilistic system. These classes of systems arise in a variety of fields and have been studied by various communities. Recursive Markov chains (RMC), and the equivalent model of probabilistic pushdown systems (pPDS), are natural models for probabilistic programs with recursive procedures [11, 8]. Quasi-birth-death (QBD) processes, which are essentially equivalent (in discrete-time) to probabilistic 1-counter automata (p1CA), are used in queueing theory and performance evaluation [20, 18]. Stochastic context-free grammars are a central model in natural language processing and are used also in biology [6], and branching processes are a classical probabilistic model with many applications, including in population genetics ([14]).

A central problem for the analysis and model checking of these systems is the computation of their associated *termination probabilities*. Computing these probabilities amounts to solving a system of fixed-point multivariate equations $x = P(x)$, where x is a (finite) vector of variables and P is a vector of polynomials with positive coefficients; such a system of equations is called a *monotone polynomial system* (MPS) because P defines a monotone operator from the nonnegative orthant to itself. Each of the above classes has the property that, given a model M in the class, we can construct in polynomial time a corresponding MPS $x = P(x)$ such that the termination probabilities of M (for various initial states) are the *least fixed point* (LFP) solution of the system, i.e., they satisfy the system, and any other nonnegative solution is at least as large in every coordinate. In general, a monotone polynomial system may not have any fixed point; consider for example $x = x + 1$. However, if it has a fixed point, then it has a least fixed point (LFP). The systems constructed from probabilistic systems as above always have a LFP, which has values in $[0, 1]$ since its coordinates give the termination probabilities.

The equations are in general nonlinear, and their LFP solution (the vector of termination probabilities) is in general irrational even when all the coefficients of the polynomials (and the numerical input data of the given probabilistic model) are rational. Hence we seek to compute the desired quantities up to a desired accuracy $\epsilon > 0$. The goal is to compute them as efficiently as possible, as a function of the encoding size of the input (the given probabilistic model, or the MPS) and the accuracy ϵ. We first review some of the relevant previous work and then describe our results.

Previous Work. An algorithm for computing the LFP of MPSs, based on Newton's method, was proposed in [11]. Given a MPS, we can first identify in polynomial time the variables that have value 0 in the LFP and remove them from the system, yielding a new so-called *cleaned* system. Then a dependency

graph between the variables is constructed, the variables and the MPS are decomposed into strongly connected components (SCCs), and Newton's method is applied bottom-up on the SCCs, starting from the all-0 vector. It was shown in [11] that, for any MPS that has a (nonnegative) solution, the decomposed variant of Newton's method converges monotonically to the LFP. Optimized variants of decomposed Newton's method have by now been implemented in several tools (see, e.g., [22, 19]), and they perform quite well in practice on many instances.

Esparza, Kiefer and Luttenberger studied in detail the rate of convergence of Newton's method on MPSs [7] (with or without decomposition). On the negative side, they showed that there are instances of MPSs $x = P(x)$ (in fact even simple RMCs), with n variables, where it takes an exponential number of iterations in the input size to get even within just one bit of precision (i.e. accuracy $1/2$). On the positive side, they showed that after some initial number k_P of iterations in a first phase, Newton's method thereafter gains bits of precision at a linear rate, meaning that $k_P + c_P \cdot i$ iterations suffice to gain i bits of precision, where both k_P and c_P depend on the input, $x = P(x)$. For strongly connected MPSs, they showed that the length, k_P, of the initial phase is upper bounded by an exponential function of the input size $|P|$, and that $c_P = 1$. For general MPSs that are not strongly connected (and for general RMCs and pPDSs), they showed that $c_P = n2^n$ suffices, but they provided no upper bound at all on k_P (and none was known prior to the present paper). Thus, they obtained no upper bounds, as a function of the size of the input, $x = P(x)$, for the number of iterations required to get to within even the first bit of precision (e.g., to estimate within $< 1/2$ the termination probability of a RMC) for general MPSs and RMCs. Proving such a general bound was left as an open problem in [7].

For special classes of probabilistic models (and MPSs) better results are now known. For the class of quasi-birth-death processes (QBDs) and the equivalent class of probabilistic 1-counter automata (p1CA), it was shown in [10] that the decomposed Newton method converges in a polynomial number of iterations in the size of the input and the bits of precision, and hence the desired termination probabilities of a given p1CA M can be computed within absolute error $\epsilon = 2^{-i}$ in a number of arithmetic operations that is polynomial in the size $|M|$ of the input and the number $i = \log(1/\epsilon)$ of bits of precision. Note that this is *not* polynomial time in the standard Turing model of complexity, because the numbers that result from the arithmetic operations in general can become exponentially long (consider n successive squarings of a number). Thus, the result of [10] shows that the termination problem for p1CAs can be solved in polynomial time in the *unit-cost exact rational arithmetic* model, a model in which arithmetic operations cost 1 time unit, regardless of how long the numbers are. It is not known exactly how powerful the unit-cost rational model is, but it is believed to be strictly more powerful than the ordinary Turing model. The question whether the termination probabilities of a p1CA (and a QBD) can be computed in polynomial time (in the standard model) was left open in [10].

Building on the results of [10] for computation of termination probabilities of p1CAs, more recently Brazdil, Kiefer and Kucera [4] showed how to do

quantitative model checking of ω-regular properties (given by a deterministic Rabin automaton) for p1CAs, i.e., compute within desired precision $\epsilon > 0$ the probability that a run of a given p1CA, M, is accepted by a given deterministic Rabin automaton, R, in time polynomial in $M, R, \log(1/\epsilon)$ in the unit-cost rational arithmetic model. The complexity in the standard Turing model was left open.

For the classes of stochastic context-free grammars, multi-type branching processes, and the related class of 1-exit RMCs, we showed recently in [9] that termination probabilities can be computed to within precision ϵ in polynomial time in the size of the input model and $\log(1/\epsilon)$ (i.e. the # of bits of precision) in the standard Turing model [9]. The algorithm is a variant of Newton's method, where the preprocessing identifies and eliminates (in P-time [11]) the variables that have value 1 in the LFP (besides the ones with value 0). Importantly, the numbers throughout the computation are not allowed to grow exponentially in length, but are always rounded down to a polynomial number of bits. The analysis then shows that the rounded Newton's algorithm still converges to the correct values (the LFP) and the number of iterations and the entire time complexity is polynomially bounded.

For general RMCs (and pPDSs) and furthermore for general MPSs, even if the LFP is in $[0, 1]^n$, there are negative results indicating that it is probably impossible to compute the termination probabilities and the LFP in polynomial time in the standard Turing model. In particular, we showed in [11] that approximating the termination probability of a RMC within *any* constant additive error $< \frac{1}{2}$, is at least as hard as the *square-root-sum* problem, a longstanding open problem that arises often in computational geometry, which is not even known to be in NP, and that it is also as hard as the more powerful problem, called PosSLP [1], which captures the essence of unit-cost rational arithmetic. Thus, if one can approximate the termination probability of a RMC in polynomial time then it is possible to simulate unit-cost rational arithmetic in polynomial time in the standard model, something which is highly unlikely.

As we mentioned at the beginning, computing termination probabilities is a key ingredient for performing other, more general analyses, including model checking [12, 8].

Our Results. We provide a thorough analysis of decomposed Newton's method and show upper bounds on its rate of convergence as a function of the input size and the desired precision, which holds for *arbitrary* monotone polynomial systems. Furthermore, we analyze a *rounded* version of the algorithm where the results along the way are not computed exactly to arbitrary precision but are rounded to a suitable number of bits (proportional to the number of iterations k of Newton's method that are performed), while ensuring that the algorithm stays well-defined and converges to the LFP. Thus, the bounds we show hold for the standard Turing model and not only the unit-cost model. Note that all the previous results on Newton's method that we mentioned, except for [9], assume that the computations are carried out in *exact* arithmetic. To carry out k iterations of Newton's method with exact arithmetic can require exponentially

many bits, as a function of k, to represent the iterates. In general, the fact that Newton's method converges with exact arithmetic does not even imply automatically that rounded Newton iterations will get anywhere close to the solution when we round to, say, only polynomially many bits of precision as a function of the number of iterations k, let alone that the same bounds on the convergence rate will continue to hold. We nevertheless show that suitable rounding works for MPSs.

In more detail, suppose that the given (cleaned) MPS $x = P(x)$ has a LFP $q^* > 0$. The decomposition into strongly connected components yields a DAG of SCCs with depth d, and we wish to compute the LFP with (absolute) error at most ϵ. Let $q^*_{\min}$ and $q^*_{\max}$ be the minimum and maximum coordinate of q^*. Then the rounded decomposed Newton method will converge to a vector $\tilde{q}$ within ϵ of the LFP, i.e., such that $\|q^* - \tilde{q}\|_\infty \leq \epsilon$ in time polynomial in the size $|P|$ of the input, $\log(1/\epsilon)$, $\log(1/q^*_{\min})$, $\log(q^*_{\max})$, and 2^d (the depth d in the exponent can be replaced by the maximum number of *nonlinear* SCCs in any path of the DAG of SCCs). We also obtain bounds on $q^*_{\min}$ and $q^*_{\max}$ in terms of $|P|$ and the number of variables n, so the overall time needed is polynomial in $|P|$, 2^n and $\log(1/\epsilon)$. We provide actually concrete expressions on the number of iterations and the number of bits needed. As we shall explain, the bounds are essentially optimal in terms of several parameters. The analysis is quite involved and builds on the previous work. It uses several results and techniques from [11, 7, 9], and develops substantial additional machinery.

We apply our results then to probabilistic 1-counter automata (p1CAs). Using our analysis for the rounded decomposed Newton method and properties of p1CAs from [10], we show that termination probabilities of a p1CA M (and QBDs) can be computed to desired precision ϵ in polynomial time in the size $|M|$ of the p1CA and $\log(1/\epsilon)$ (the bits of precision) in the standard Turing model of computation, thus solving the open problem of [10].

Furthermore, combining with the results of [4] and [12], we show that one can do quantitative model checking of ω-regular properties for p1CAs in polynomial time in the standard Turing model, i.e., we can compute to desired precision ϵ the probability that a run of a given p1CA M satisfies an ω-regular property in time polynomial in $|M|$ and $\log(1/\epsilon)$ (and exponential in the property if it is given for example as a non-deterministic Büchi automaton or polynomial if it is given as a deterministic Rabin automaton).

The rest of the paper is organized as follows. In Section 2 we give basic definitions and background. In Section 3 we consider strongly-connected MPS, and in Section 4 general MPS. Section 5 analyzes p1CAs. Most proofs are omitted (see the full version [21]).

2 Definitions and Background

We first recall basic definitions about MPSs from [11]. A *monotone polynomial system of equations* (MPS) consists of a system of n equations in n variables, $x = (x_1, \ldots, x_n)$, the equations are of the form $x_i = P_i(x)$, $i = 1, \ldots, n$, such

that $P_i(x)$ is a multivariate polynomial in the variables x, and such that the monomial coefficients and constant term of $P_i(x)$ are all non-negative. More precisely, for $\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_n) \in \mathbb{N}^n$, we use the notation x^α to denote the monomial $x_1^{\alpha_1} x_2^{\alpha_2} \ldots x_n^{\alpha_n}$. (Note that by definition $x^{(0,\ldots,0)} = 1$.) Then for each polynomial $P_i(x)$, $i = 1, \ldots, n$, there is some *finite* subset of $\mathbb{N}^n$, denoted $\mathcal{C}_i$, and for each $\alpha \in \mathcal{C}_i$, there is a positive (rational) coefficient $c_{i,\alpha} > 0$, such that $P_i(x) \equiv \sum_{\alpha \in \mathcal{C}_i} c_{i,\alpha} x^\alpha$.

For computational purposes, we assume each polynomial $P_i(x)$ has rational coefficients[1], and that it is encoded succinctly by specifying the list of pairs $\langle (c_{i,\alpha}, \alpha) \mid \alpha \in \mathcal{C}_i \rangle$, where each rational coefficient $c_{i,\alpha}$ is represented by giving its numerator and denominator in binary, and each integer vector α is represented in *sparse representation*, by only listing its non-zero coordinates, $i_1, \ldots, i_k$, by using a list $\langle (i_1, \alpha)_{i_1}), \ldots, (i_k, \alpha_{i_k}) \rangle$, giving each integer α_{i_j} in binary. (Proposition 1 below, from [11, 9], shows that using such a sparse representation does not entail any extra computational cost.)

We use vector notation, using $x = P(x)$ to denote the entire MPS. We use $|P|$ to denote the encoding size (in bits) of the MPS $x = P(x)$ having rational coefficients, using the succinct representation just described.

Let $\mathbb{R}_{\geq 0}$ denote the non-negative real numbers. Then $P(x)$ defines a monotone operator on the non-negative orthant $\mathbb{R}^n_{\geq 0}$. In other words, $P : \mathbb{R}^n_{\geq 0} \to \mathbb{R}^n_{\geq 0}$, and if $\mathbf{0} \leq a \leq b$, then $P(a) \leq P(b)$. In general, an MPS need not have any real-valued solution: consider $x = x + 1$. However, because of monotonicity of $P(x)$, if there exists a solution $a \in \mathbb{R}^n_{\geq 0}$ such that $a = P(a)$, then there exists a *least fixed point* (LFP) solution $q^* \in \mathbb{R}^n_{\geq 0}$ such that $q^* = P(q^*)$, and such that $q^* \leq a$ for all solutions $a \in \mathbb{R}^n_{\geq 0}$. Indeed, if for $z \in \mathbb{R}^n$ we define $P^0(z) = z$, and define $P^{k+1}(z) = P(P^k(z))$, for all $k \geq 0$, then (as shown in [11]) value iteration starting at the all-0 vector $\mathbf{0}$ converges monotonically to q^*: in other words $\forall k \geq 0\ P^k(\mathbf{0}) \leq P^{k+1}(\mathbf{0})$, and $\lim_{k\to\infty} P^k(\mathbf{0}) = q^*$.[2]

Unfortunately, standard value iteration $P^k(0), k \to \infty$, can converge very slowly to q^*, even for a fixed MPS with 1 variable, even when $q^* = 1$; specifically, $x = (1/2)x^2 + 1/2$ already exhibits exponentially slow convergence to its LFP $q^* = 1$

[1] Although we also reason about MPSs with positive real-valued coefficients in our proofs.

[2] Indeed, even if an MPS does not have a *finite* LFP solution $q^* \in \mathbb{R}^n_{\geq 0}$, it always does have an LFP solution *over the extended non-negative reals*. Namely, we can define the LFP of any MPS, $x = P(x)$, to be the vector $q^* \in \overline{\mathbb{R}}^n_{\geq 0}$ over $\overline{\mathbb{R}}_{\geq 0} = (\mathbb{R}_{\geq 0} \cup \{+\infty\})$, given by $q^* := \lim_{k\to\infty} P^k(\mathbf{0})$. In general, it is PosSLP-hard to decide whether a given MPS has a finite LFP. (This follows easily from results in [11], although it is not stated there: is was shown there it is PosSLP-hard to decide if $q_1^* \geq 1$ in an MPS with finite LFP $q^* \in \mathbb{R}^n_{\geq 0}$. Then just add a variable x_0, and an equation $x_0 = x_0 x_1 + 1$ to the MPS. In the new MPS, $q_0^* = +\infty$ if and only if $q_1^* \geq 1$.) However, various classes of MPSs, including those whose LFP corresponds to termination probabilities of various recursive probabilistic systems do have a finite LFP. Thus in this paper we will only consider LFP computation for MPSs that have a finite LFP $q^* \in \mathbb{R}^n_{\geq 0}$. So when we say "$x = P(x)$ is an MPS with LFP solution q^*", we mean $q^* \in \mathbb{R}^n_{\geq 0}$, unless specified otherwise.

([11]). It was shown in [11] that a decomposed variant of Newton's method also converges monotonically to q^* for an MPS with LFP solution q^*. More recently, in [9], a version of Newton's method with suitable rounding between iterations was studied. Rounding is necessary if one wishes to consider the complexity of Newton's method in the standard (Turing) model of computation, which does not allow unit-cost arithmetic operations on arbitrarily large numbers. In this paper we will apply a version of Newton's method to MPSs which uses both rounding and decomposition. Before describing it, we need some further background.

An MPS, $x = P(x)$, is said to be in *simple normal form* (SNF) if for every $i = 1, \ldots, n$, the polynomial $P_i(x)$ has one of two forms: (1) $\underline{\text{Form}_*}$: $P_i(x) \equiv x_j x_k$ is simply a quadratic monomial; or (2) $\underline{\text{Form}_+}$: $P_i(x)$ is a *linear* expression $\sum_{j \in \mathcal{C}_i} p_{i,j} x_j + p_{i,0}$, for some rational non-negative coefficients $p_{i,j}$ and $p_{i,0}$, and some index set $\mathcal{C}_i \subseteq \{1, \ldots, n\}$. In particular, in any MPS in SNF form every polynomial $P_i(x)$ has multivariate degree bounded by at most 2 in the variables x. We will call such MPSs **quadratic MPSs**.

As shown in [11, 9], it is easy to convert any MPS to SNF form, by adding auxiliary variables and equations:

Proposition 1. *(Propos. 7.3 [11], and Propos. 2.1 of [9]) Every MPS, $x = P(x)$, with LFP q^*, can be transformed in P-time to an "*equivalent*" quadratic MPS $y = Q(y)$ in SNF form, such that $|Q| \in O(|P|)$. More precisely, the variables x are a subset of the variables y, and $y = Q(y)$ has LFP p^* iff $x = P(x)$ has LFP q^*, and projecting p^* onto the x variables yields q^*.*

Furthermore, for any MPS, $x = P(x)$, we can in P-time find and remove any variables x_i, such that the LFP solution has $q_i^* = 0$.[3]

Proposition 2. *(Proposition 7.4 of [11]) There is a P-time algorithm that, given any MPS[3], $x = P(x)$, over n variables, determines for each $i \in \{1, \ldots, n\}$ whether $q_i^* = 0$.*

Thus, for every MPS, we can detect in P-time all the variables x_j such that $q_j^* = 0$, remove their equation $x_j = P_j(x)$, and set the variable x_j to 0 on the RHS of the remaining equations. We obtain as a result a ***cleaned*** MPS, $x' = Q(x')$, which has an LFP $q^* > 0$.

Applying Propositions 1 and 2, ***we assume wlog in the rest of this paper that every MPS is a cleaned quadratic MPS, with LFP*** $q^* > 0$.[4]

In order to describe *decomposed* Newton's method, for a *cleaned* MPS, $x = P(x)$ we need to define the *dependency graph*, $G_P = (V, E)$, of the MPS. The nodes V of G_P are the remaining variables x_i, and the edges are defined as follows: $(x_i, x_j) \in E$ if and only if x_j appears in some monomial in $P_i(x)$ that has a positive coefficient.

[3] This proposition holds regardless whether the LFP q^* is *finite* or is over the *extended non-negative reals*, $\overline{\mathbb{R}}_{\geq 0}$. Such an extended LFP exists for any MPS. See footnote 2.

[4] For compatibility when quoting prior work, it will sometimes be convenient to assume quadratic MPSs, rather than the more restricted SNF form MPSs.

We shall decompose the cleaned system of equation $x = P(x)$, into strongly connected components (SCCs), using the dependency graph G_P of variables, and we shall apply Newton's method separately on each SCC "bottom-up".

We first recall basic definitions for (a rounded down version of) Newton's method applied to MPSs. For an MPS, $x = P(x)$, with n variables, we define $B(x) = P'(x)$ to be the $n \times n$ Jacobian matrix of partial derivatives of $P(x)$. In other words, $B(x)_{i,j} = \frac{\partial P_i(x)}{\partial x_j}$. For a vector $z \in \mathbb{R}^n$, assuming that the matrix $(I - B(z))$ is non-singular, a single iteration of *Newton's method* (**NM**) on $x = P(x)$ at z is defined via the following operator:

$$\mathcal{N}_P(z) := z + (I - B(z))^{-1}(P(z) - z) \tag{1}$$

Let us now recall from [9] the rounded down Newton's method, with parameter h, applied to an MPS:

Definition 1. Rounded-Down Newton's Method *(***R-NM***) , with rounding parameter h.) Given an MPS, $x = P(x)$, with LFP q^*, where $\mathbf{0} < q^*$, in the* rounded down Newton's method *(R-NM) with integer rounding parameter $h > 0$, we compute a sequence of iteration vectors $x^{[k]}$, where the initial starting vector is $x^{[0]} := \mathbf{0}$, and such that for each $k \geq 0$, given $x^{[k]}$, we compute $x^{[k+1]}$ as follows:*

1. *First, compute $x^{\{k+1\}} := \mathcal{N}_P(x^{[k]})$, where the Newton iteration operator $\mathcal{N}_P(x)$ was defined in equation (1). (Of course we need to show that all such Newton iterations are defined.)*
2. *For each coordinate $i = 1, \ldots, n$, set $x_i^{[k+1]}$ to be equal to the maximum (non-negative) multiple of 2^{-h} which is $\leq \max(x_i^{\{k+1\}}, 0)$. (In other words, round down $x^{\{k+1\}}$ to the nearest multiple of 2^{-h}, while making sure that the result is non-negative.)*

Now we describe the ***Rounded-down Decomposed Newton's Method*** (**R-DNM**) applied to an MPS, $x = P(x)$, with real-valued LFP $q^* \geq 0$. Firstly, we use Proposition 2 to remove 0 variables, and thus we can assume we are given a cleaned MPS, $x = P(x)$, with real-valued LFP $q^* > 0$.

Let H_P be the DAG of SCC's of the dependency graph G_P. We work bottom-up in H_P, starting at bottom SCCs. For each SCC, S, suppose its corresponding equations are $x_S = P_S(x_S, x_{D(S)})$, where $D(S)$ denotes the union of the variables in "lower" SCCs, below S, on which S depends. In other words, a variable $x_j \in D(S)$ iff there is some variable $x_i \in S$ such that there is directed path in G_p from x_i to x_j. If the system $x_S = P_S(x_S, q^*_{D(S)})$ is a linear system (in x_S), we call S a *linear SCC*, otherwise S is a *nonlinear SCC*. Assume we have already calculated (using R-DNM) an approximation $\tilde{q}_{D(S)}$ to the LFP solution $q^*_{D(S)}$ for these lower SCCs. We plug in $\tilde{q}_{D(S)}$ into the equations for S, obtaining the equation system $x_S = P_S(x_S, \tilde{q}_{D(S)})$. We denote the actual LFP solution of this new equation system by q'_S. (Note that q'_S is not necessarily equal to q^*_S, because $\tilde{q}_{D(S)}$ is only an approximation of $q^*_{D(S)}$.)

If S is a nonlinear SCC, we apply a chosen number g of iterations of R-NM on the system $x_S = P_S(x_S, \tilde{q}_{D(S)})$ to obtain an approximation $\tilde{q}_S$ of q'_S; if S is linear then we just apply 1 iteration of R-NM, i.e., we solve the linear system and round down the solution. We of course want to make sure our approximations are such that $\|q^*_S - \tilde{q}_S\|_\infty \leq \epsilon$, for all SCCs S, and for the desired additive error $\epsilon > 0$. We shall establish upper bounds on the number of iterations g, and on the rounding parameter h, needed in R-DNM for this to hold, as a function of various parameters: the input size $|P|$ and the number n of variables; the *nonlinear depth* f of P, which is defined as the maximum, over all paths of the DAG H_P of SCCs, of the number of nonlinear SCCs on the path; and the maximum and minimum coordinates of the LFP.

Bounds on the Size of LFPs for an MPS. For a positive vector $v > 0$, we use $v_{\min} = \min_i v_i$ to denote its minimum coordinate, and we use $v_{\max} = \max_i v_i$ to denote its maximum coordinate. Slightly overloading notation, for an MPS, $x = P(x)$, we shall use $c_{\min}$ to denote the minimum value of all positive monomial coefficients and all positive constant terms in $P(x)$. Note that $c_{\min}$ also serves as a lower bound for all positive constants and coefficients for entries of the Jacobian matrix $B(x)$, since $B(x)_{ij} = \frac{\partial P_i(x)}{\partial x_j}$.

We prove the following Theorem in the full paper [21], establishing bounds on the maximum and minimum coordinates of the LFP q^* of an MPS $x = P(x)$.

Theorem 1. *If $x = P(x)$ is a quadratic MPS in n variables, with LFP $q^* > 0$, and where $P(x)$ has rational coefficients and total encoding size $|P|$ bits, then*

1. $q^*_{\min} \geq 2^{-|P|(2^n-1)}$, and
2. $q^*_{\max} \leq 2^{2(n+1)(|P|+2(n+1)\log(2n+2))\cdot 5^n}$.

How Good Are Our Upper Bounds? The full paper [21] discusses how good our upper bounds on R-DNM are, and in what senses they are optimal, in light of the convergence rate of Newton's method on known bad examples ([7]), and considerations relating to the size of $q^*_{\min}$ and $q^*_{\max}$. In this way, our upper bounds can be seen to be essentially optimal in several parameters, including the depth of SCCs in the dependency graph of the MPS, and in terms of $\log \frac{1}{\epsilon}$.

3 Strongly Connected Monotone Polynomial Systems

Theorem 2. *Let $P(x, y)$ be an n-vector of monotone polynomials with degree ≤ 2 in variables which are coordinates of the n-vector x and the m-vector y, where $n \geq 1$ and $m \geq 1$.*

Given non-negative m-vectors y_1 and y_2 such that $0 < y_1 \leq \mathbf{1}$ and $0 \leq y_2 \leq y_1$, let $P_1(x) \equiv P(x, y_1)$ and $P_2(x) \equiv P(x, y_2)$. Suppose that $x = P_1(x)$ is a strongly-connected MPS with LFP solution $0 < q^*_1 \leq \mathbf{1}$.

Let $\alpha = \min\{1, c_{\min}\}\min\{y_{\min}, \frac{1}{2}q^*_{\min}\}$, where $c_{\min}$ is the smallest non-zero constant or coefficient of any monomial in $P(x, y)$, where $y_{\min}$ is the minimum coordinate of y_1, and finally where $q^*_{\min}$ is the minimum coordinate of q^*_1. Then:

1. *The LFP solution of the MPS $x = P_2(x)$ is q_2^* with $0 \leq q_2^* \leq q_1^*$, and*

$$\|q_1^* - q_2^*\|_\infty \leq \sqrt{4n\alpha^{-(3n+1)}\|P(\mathbf{1}, \mathbf{1})\|_\infty \|y_1 - y_2\|_\infty}$$

Furthermore, if $x = P_1(x)$ is a linear system, then:

$$\|q_1^* - q_2^*\|_\infty \leq 2n\alpha^{-(n+2)}\|P(\mathbf{1}, \mathbf{1})\|_\infty \|y_1 - y_2\|_\infty$$

2. *Moreover, for every $0 < \epsilon < 1$, if we use $g \geq h - 1$ iterations of rounded down Newton's method with parameter*

$$h \geq \lceil 2 + n \log \frac{1}{\alpha} + \log \frac{1}{\epsilon} \rceil$$

applied to the MPS, $x = P_2(x)$, starting at $x^{[0]} := \mathbf{0}$, to approximate q_2^*, then the iterations are all defined, and $\|q_2^* - x^{[g]}\|_\infty \leq \epsilon$.

Theorem 2 and its proof are at the heart of this paper, but unfortunately the proof is quite involved, and we have no room to include it. The proof is in [21]. The following easy corollary is also proved in [21].

Corollary 1. *Let $x = P(x)$ be a strongly connected MPS with n variables, and with LFP q^* where $0 < q^* \leq 1$. Let $\alpha = \min\{1, c_{\min}\}\frac{1}{2}q^*_{\min}$, where $c_{\min}$ is the smallest non-zero constant or coefficient of any monomial in $P(x)$.*

Then for all $0 < \epsilon < 1$, if we use $g \geq h-1$ iterations of R-NM with parameter $h \geq \lceil 2 + n \log \frac{1}{\alpha} + \log \frac{1}{\epsilon} \rceil$ applied to the MPS, $x = P(x)$, starting at $x^{[0]} := \mathbf{0}$, then the iterations are all defined, and $\|q^* - x^{[g]}\|_\infty \leq \epsilon$.

4 General Monotone Polynomial Systems

In this section, we use the rounded-down decomposed Newton's method (R-DNM), to compute the LFP q^* of general MPSs. First we consider the case where $0 < q^* \leq 1$:

Theorem 3. *For all ϵ, where $0 < \epsilon < 1$, if $x = P(x)$ is an MPS with LFP solution $0 < q^* \leq 1$, with $q^*_{\min} = \min_i q_i^*$, and the minimum non-zero coefficient or constant in $P(x)$ is $c_{\min}$, then rounded down decomposed Newton's method (R-DNM) with parameter*

$$h \geq \left\lceil 3 + 2^f \cdot \left(\log(\frac{1}{\epsilon}) + d \cdot (\log(\alpha^{-(4n+1)}) + \log(16n) + \log(\|P(\mathbf{1})\|_\infty)) \right) \right\rceil$$

using $g \geq h - 1$ iterations for every nonlinear SCC (and 1 iteration for linear SCC), gives an approximation $\tilde{q}$ to q^* with $\tilde{q} \leq q^*$ and such that $\|q^* - \tilde{q}\|_\infty \leq \epsilon$.

Here d denotes the maximum depth of SCCs in the DAG H_P of SCCs of the MPS $x = P(x)$, f is the nonlinear depth, and $\alpha = \min\{1, c_{\min}\} \cdot \frac{1}{2}q^*_{\min}$.

Before proving the theorem, let us note that we can obtain worst-case expressions for the needed number of iterations $g = h-1$, and the needed rounding parameter h, in terms of only $f \le d \le n \le |P|$, and ϵ, by noting that $\log(\|P(\mathbf{1})\|_\infty) \le |P|$, and by appealing to Theorem 1 to remove references to $q^*_{\min}$ in the bounds. Noting that $c_{\min} \ge 2^{-|P|}$, these tell us that $\min\{1, c_{\min}\}\frac{1}{2}q^*_{\min} \ge 2^{-|P|2^n-1}$. Substituting, we obtain that any:

$$g \ge \left\lceil 2 + 2^f \cdot (\log(\frac{1}{\epsilon}) + d \cdot (|P|2^n(4n+1) + (4n+1) + \log(16n) + |P|)) \right\rceil \quad (2)$$

iterations suffice in the worst case, with rounding parameter $h = g + 1$. Thus, for $i = \log(1/\epsilon)$ bits of precision, $g = k_P + c_P \cdot i$ iterations suffice, where $c_P = 2^f$ and $k_P = O(2^f 2^n nd|P|)$, with tame constants in the big-O.

Proof (of Theorem 3). For every SCC S, its *height* h_S (resp. *nonlinear height* f_S) is the maximum over all paths of the DAG H_P of SCCs starting at S, of the number of SCCs (resp. nonlinear SCCs) on the path. We show by induction on the height h_S of each SCC S that $\|q^*_S - \tilde{q}_S\|_\infty \le \beta^{h_S}\delta^{2^{-f_S}}$ where $\beta = 16n\alpha^{-(3n+1)}\|P(\mathbf{1})\|_\infty$ and $\delta = (\frac{\epsilon}{\beta^d})^{2^f}$. Note that since $n \ge 1$, $\epsilon < 1$, and $\alpha \le c_{\min}$, we have $\beta \ge 1$ and $\delta \le 1$, and thus also $\delta \le \sqrt{\delta}$.

Let us first check that this would imply the theorem. For all SCCs, S, we have $1 \le h_S \le d$ and $0 \le f_S \le f$, and thus $\|q^*_S - \tilde{q}_S\|_\infty \le \beta^{h_S}\delta^{2^{-f_S}} \le \beta^d\delta^{2^{-f}} = \beta^d(\frac{\epsilon}{\beta^d}) = \epsilon$.

We note that h is related to δ by the following:

$$h \ge 2 + n\log\frac{1}{\alpha} + \log\frac{2}{\delta} \quad (3)$$

This is because $\log\frac{2}{\delta} = 1 + \log\frac{1}{\delta} = 1 + 2^f(\log\frac{1}{\epsilon} + d\log\beta) = 1 + 2^f(\log(\frac{1}{\epsilon}) + d\log(16n\alpha^{-3n+1}\|P(\mathbf{1})\|_\infty))$. Note that (3) implies that this inequality holds also for any subsystem of $x = P(x)$ induced by a SCC S and its successors $D(S)$ because the parameters n and $1/\alpha$ for a subsystem are no larger than those for the whole system.

We now prove by induction on h_S that $\|q^*_S - \tilde{q}_S\|_\infty \le \beta^{h_S}\delta^{2^{-f_S}}$.

In the base case, $h_S = 1$, we have a strongly connected MPS $x_S = P_S(x)$. If S is linear, we solve the linear system exactly and then round down to a multiple of 2^{-h}. Then $f_S = 0$, and we have to show $\|q^*_S - \tilde{q}_S\|_\infty \le \beta^{h_S}\delta^{2^{-f_S}} = \beta\delta$. But $\|q^*_S - \tilde{q}_S\|_\infty \le 2^{-h} \le \frac{\delta}{2} \le \beta\delta$.

For the base case where S in non-linear, equation (3) and Corollary 1 imply that $\|q^*_S - \tilde{q}_S\|_\infty \le \frac{\delta}{2}$, which implies the claim since $\delta \le 1$ and $\beta \ge 1$, hence $\frac{\delta}{2} \le \beta^{h_S}\delta^{2^{-f_S}} = \beta^1\delta^{2^{-1}}$.

Inductively, consider an SCC S with $h_S > 1$. Then S depends only on SCCs with height at most $h_S - 1$. If S is linear, it depends on SCCs of nonlinear depth at most $f_{D(S)} = f_S$, whereas if S is non-linear, it depends on SCCs of nonlinear depth at most $f_{D(S)} = f_S - 1$. We can assume by inductive hypothesis that $\|q^*_{D(S)} - \tilde{q}_{D(S)}\|_\infty \le \beta^{h_S-1}\delta^{2^{-f_{D(S)}}}$. Take q'_S to be the LFP of $x_S = P_S(x_S, \tilde{q}_{D(S)})$.

Suppose $x_S = P_S(x_S, q^*_{D(S)})$ is linear in x_S. Then Theorem 2 with $y_1 := q^*_{D(S)}$ and $y_2 := \tilde{q}_{D(S)}$, yields

$$\|q^*_S - q'_S\|_\infty \leq 2n_S\alpha^{-(n_S+2)}\|P(\mathbf{1},\mathbf{1})\|_\infty\|q^*_{D(S)} - \tilde{q}_{D(S)}\|_\infty$$

But $2n_S\alpha^{-(n_S+2)}\|P(\mathbf{1},\mathbf{1})\|_\infty \leq \frac{\beta}{2}$, so $\|q^*_S - q'_S\|_\infty \leq \frac{\beta}{2}\|q^*_{D(S)} - \tilde{q}_{D(S)}\|_\infty \leq \frac{\beta}{2}\beta^{h_S-1}\delta^{2^{-f_S}} = \frac{1}{2}\beta^{h_S}\delta^{2^{-f_S}}$. Since $\|q'_S - \tilde{q}_S\|_\infty \leq 2^{-h} \leq \frac{\delta}{2} \leq \frac{1}{2}\beta^{h_S}\delta^{2^{-f_S}}$, it follows that $\|q^*_S - \tilde{q}_S\|_\infty \leq \beta^{h_S}\delta^{2^{-f_S}}$.

Suppose that $x_S = P_S(x_S, q^*_{D(S)})$ is non-linear in x_S. Theorem 2, with $y_1 := q^*_{D(S)}$ and $y_2 := \tilde{q}_{D(S)}$, yields that

$$\|q^*_S - q'_S\|_\infty \leq \sqrt{4n\alpha^{-(3n+1)}\|P(\mathbf{1})\|_\infty\|q^*_{D(S)} - (\tilde{q})_{D(S)}\|_\infty} \tag{4}$$

Note that the α from Theorem 2 is indeed the same or better (i.e., bigger) than the α in this Theorem, because $y_{\min} = (q^*_{D(S)})_{\min} \geq q^*_{\min}$ and $(q^*_S)_{\min} \geq q^*_{\min}$. Rewriting (4) in terms of β, we have $\|q^*_S - q'_S\|_\infty \leq \sqrt{\frac{1}{4}\beta\|q^*_{D(S)} - (\tilde{q})_{D(S)}\|_\infty}$. By inductive assumption, $\|q^*_{D(S)} - \tilde{q}_{D(S)}\|_\infty \leq \beta^{h_S-1}\delta^{2^{-f_S+1}}$, and thus $\|q^*_S - q'_S\|_\infty \leq \sqrt{\frac{1}{4}\beta^{h_S}\delta^{2^{1-f_S}}} \leq \frac{1}{2}\beta^{h_S}\delta^{2^{-f_S}}$. Thus to show that the inductive hypothesis holds also for SCC S, it suffices to show that for the approximation $\tilde{q}_S$ we have $\|q'_S - \tilde{q}_S\|_\infty \leq \frac{1}{2}\beta^{h_S}\delta^{2^{-f_S}}$. But $\beta \geq 1$, $h_S \geq 1$, $2^{-f_S} \leq 1$ and $\delta \leq 1$, so $\frac{1}{2}\delta \leq \frac{1}{2}\beta^{h_S}\delta^{2^{-f_S}}$, so it suffices to show that $\|q'_S - \tilde{q}_S\|_\infty \leq \frac{1}{2}\delta$. Part 2 of Theorem 2 tells us that we will have $\|q'_S - \tilde{q}_S\|_\infty \leq \frac{1}{2}\delta$ if $g \geq h-1$ and $h \geq 2+n\log\frac{1}{\alpha}+\log\frac{2}{\delta}$. But we have already established this in equation (3), hence the claim follows.

□

Next, we want to generalize Theorem 3 to arbitrary MPSs that have an LFP, $q^* > 0$, without the restriction that $0 < q^* \leq 1$. The next Lemma allows us to establish this by a suitable "rescaling" of any MPS which has an LFP $q^* > 0$. If $x = P(x)$ is a MPS and $c > 0$, we can consider the MPS $x = \frac{1}{c}P(cx)$.

Lemma 1. *Let $x = P(x)$ be a MPS with LFP solution q^*, and with Jacobian $B(x)$, and recall that for $z \geq 0$, $\mathcal{N}_P(z) := z + (I - B(z))^{-1}(P(z) - z)$ denotes the Newton operator applied at z on $x = P(x)$. Then:*

(i) The LFP solution of $x = \frac{1}{c}P(cx)$ is $\frac{1}{c}q^*$.
(ii) The Jacobian of $\frac{1}{c}P(cx)$ is $B(cx)$.
(iii) A Newton iteration of the "rescaled" MPS, $x = \frac{1}{c}P(cx)$, applied to the vector z is given by $\frac{1}{c}\mathcal{N}_P(cz)$.

Proof. From [11], we know that the value iteration sequence $P(0)$, $P(P(0))$, $P(P(P(0)))\ldots P^k(0)$ converges to q^*. Now note that for the MPS $x = \frac{1}{c}P(cx)$, the value iteration sequence is $\frac{1}{c}P(0)$, $\frac{1}{c}P(c\frac{1}{c}P(0)) = \frac{1}{c}P(P(0))$, $\frac{1}{c}P(P(P(0)))$... which thus converges to $\frac{1}{c}q^*$. This establishes (i).

For (ii), note that, by the chain rule in multivariate calculus (see, e.g., [2] Section 12.10), the Jacobian of $P(cx)$ is $cB(cx)$. Now (iii) follows because:

$$z + (I - B(cz))^{-1}(\frac{1}{c}P(cz) - z) = \frac{1}{c}(cz + (I - B(cz))^{-1}(P(cz) - cz)) = \frac{1}{c}\mathcal{N}_P(cz).$$

□

We use Lemma 1 to generalise Theorem 3 to MPSs with LFP q^*, where q^* does not satisfy $q^* \leq 1$.

Theorem 4. *If $x = Q(x)$ is an MPS with n variables, with LFP solution $q^* > 0$, if $c'_{\min}$ is the least positive coefficient of any monomial in $Q(x)$, then R-DNM with rounding parameter h', and using g' iterations per nonlinear SCC (and one for linear), gives an approximation $\tilde{q}$ such that $\|q^* - \tilde{q}\|_\infty \leq \epsilon'$, where*

$$g' = 2 + \lceil\ 2^f \cdot (\log(\frac{1}{\epsilon'}) + d \cdot (2u + \log(\alpha'^{-(4n+1)}) + \log(16n) + \log(\|Q(\mathbf{1})\|_\infty))\ )\ \rceil$$

and $h' = g' + 1 - u$, where $u = \max\{0, \lceil \log q^*_{\max} \rceil\}$, d is the maximum depth of SCCs in the DAG H_Q of SCCs of $x = Q(x)$, f is the nonlinear depth, and $\alpha' = 2^{-2u} \min\{1, c'_{\min}\} \min\{1, \frac{1}{2} q^*_{\min}\}$.

We can again obtain worst-case expressions for the needed number of iterations g', and the needed rounding parameter h', in terms of only $f \leq d \leq n \leq |Q|$, and ϵ', by noting that $\log(\|Q(\mathbf{1})\|_\infty) \leq |Q|$ and by appealing to Theorem 1 to remove references to $q^*_{\min}$ and $q^*_{\max}$ in the bounds. Substituting and simplifying we get that to guarantee additive error at most ϵ', i.e. for $i = \log(1/\epsilon')$ bits of precision, it suffices in the worst-case to apply $g' = k_Q + c_Q \cdot i$ iterations of R-DNM with rounding parameter $h' = g' + 1$ (which is more accurate rounding than $h' = g' + 1 - u$), where $c_Q = 2^f$, and $k_Q = O(2^f 5^n n^2 d(|Q| + n \log n))$ (and we can calculate precise, tame, constants for the big-O expression).

Corollary 2. *If $x = P(x)$ is an MPS with LFP solution q^* with $0 < q^*_{\min} \leq q^*_i \leq q^*_{\max}$ for all i, with the least coefficient of any monomial in $P(x)$, $c_{\min}$, with f the nonlinear depth of the DAG of SCCs of $x = P(x)$ and with encoding size $|P|$ bits, we can compute an approximation $\tilde{q}$ to q^* with $\|q^* - \tilde{q}\|_\infty \leq \epsilon$, for any given $0 < \epsilon \leq 1$, in time polynomial in $|P|$,2^f, $\log \frac{1}{\epsilon}$,$\log \frac{1}{q^*_{\min}}$ and $\log q^*_{\max}$.*

Proof. After preprocessing to remove all variables x_i with $q^*_i = 0$, which takes P-time in $|P|$, we use R-DNM as specified in Theorem 4. Calculating a Newton iterate at z is just a matter of solving a matrix equation and if the coordinates of z are multiples of 2^{-h} this can be done in time polynomial in $|P|$ and h. Theorem 4 tells us that the number of iterations and h are polynomial in 2^f, $\log \frac{1}{\epsilon}$, $\log \frac{1}{q^*_{\min}}$, $\log q^*_{\max}$, n, $\log \frac{1}{c_{\min}}$ and $\log \|P(\mathbf{1})\|_\infty$. The last three of these are bounded by $|P|$. Together, these give the corollary. □

5 MPSs and Probabilistic 1-Counter Automata

A **probabilistic 1-counter automaton** (p1CA), M, is a 3-tuple $M = (V, \delta, \delta_0)$ where V is a finite set of *control states* and $\delta \subseteq V \times \mathbb{R}_{>0} \times \{-1, 0, 1\} \times V$ and $\delta_0 \subseteq V \times \mathbb{R}_{>0} \times \{0, 1\} \times V$ are *transition relations*. The transition relation δ is enabled when the counter is nonzero, and the transition relation δ_0 is enabled

when it is zero. For example, a transition of the form, $(u, p, -1, v) \in \delta$, says that if the counter value is positive, and we are currently in control state u, then with probability p we move in the next step to control state v and we decrement the counter by 1. A p1CA defines in the obvious way an underlying countably infinite-state (labeled) Markov chain, whose set of configurations (states) are pairs $(v, n) \in V \times \mathbb{N}$. A *run* (or *trajectory*, or *sample path*), starting at initial state (v_0, n_0) is defined in the usual way, as a sequence of configurations $(v_0, n_0), (v_1, n_1), (v_2, n_2), \ldots$ that is consistent with the transition relations of M.

As explained in [10], p1CAs are in a precise sense equivalent to discrete-time *quasi-birth-death processes* (QBDs), and to *1-box recursive Markov chains*.

Quantities that play a central role for the analysis of QBDs and p1CAs (both for transient analyses and steady-state analyses, as well as for model checking) are their *termination probabilities* (also known as their *G-matrix* in the QBD literature, see, e.g., [18, 3, 10]). These are defined as the probabilities, $q^*_{u,v}$, of hitting counter value 0 for the first time in control state $v \in V$, when starting in configuration $(u, 1)$.

Corresponding to the termination probabilities of every QBD or p1CA is a special kind of MPS, $x = P(x)$, whose LFP solution q^* gives the termination probabilities of the p1CA. The MPSs corresponding to p1CAs have the following special structure. For each pair of control states $u, v \in V$ of the p1CA, there is a variable x_{uv}. The equation for each variable x_{uv} has the following form:

$$x_{uv} = p^{(-1)}_{uv} + \left(\sum_{w \in V} p^{(0)}_{uw} x_{wv} \right) + \sum_{y \in V} p^{(1)}_{uy} \sum_{z \in V} x_{yz} x_{zv} \tag{5}$$

where for all states $u, v \in V$, and $j \in \{-1, 0, 1\}$, the coefficients $p^{(j)}_{uv}$ are non-negative transition probabilities of the p1CA, and such that for all states $u \in V$, $\sum_{j \in \{-1,0,1\}} \sum_{v \in V} p^{(j)}_{uv} \leq 1$. We can of course clean up this MPS in P-time (by Proposition 2), to remove all variables x_{uv} for which $q^*_{u,v} = 0$. In what follows, we assume this has been done, and thus that for the remaining variables $0 < q^* \leq 1$.

In [10], the decomposed Newton's method (DNM) is used *with exact arithmetic* in order to approximate the LFP for p1CAs using polynomially many arithmetic operations, i.e., in polynomial time in the *unit-cost arithmetic model of computation*. However [10] did not establish any result about the rounded down version of DNM, and thus no results on the time required in the standard Turing model of computation. We establish instead results about R-DNM applied to the MPSs arising from p1CAs, in order to turn this method into a P-time algorithm in the standard model of computation.

It was shown in [10] that in any path through the DAG of SCCs of the dependency graph for the MPS associated with a p1CA, M, there is at most one non-linear SCC, i.e. the nonlinear depth is ≤ 1. Also, [10] obtained a lower bound on $q^*_{\min}$, the smallest positive termination probability. Namely, if $c_{\min}$ denotes the smallest positive transition probability of a p1CA, M, and thus also the smallest positive constant or coefficient of any monomial in the corresponding MPS, $x = P(x)$, they showed:

Lemma 2. *(Corollary 6 from [10])* $q^*_{\min} \geq c^{r^3}_{\min}$, *where* r *is the number of control states of the p1CA.*

They used these results to bound the *condition number* of the Jacobian matrix for each of the linear SCCs, and to thereby show that one can approximate q^* in polynomially many arithmetic operations using decomposed Newton's method. Here, we get a stronger result, placing the problem of computing termination probabilities for p1CA in P-time in the standard Turing model, using the results from this paper:

Theorem 5. *Let* $x = P(x)$ *be the MPS associated with p1CA,* M*, let* r *denote the number of control states of* M*, and let* m *denote the maximum number of bits required to represent the numerator and denominator of any positive rational transition probability in* M*.*

Apply R-DNM, including rounding down linear SCCs, to the MPS $x = P(x)$*, using rounding parameter* $h := 8mr^7 + 2mr^5 + 9r^2 + 3 + \lceil 2 \log \frac{1}{\epsilon} \rceil$ *and such that for each non-linear SCC we perform* $g = h - 1$ *iterations, whereas for each linear SCC we only perform* 1 *R-NM iteration.*

This algorithm computes an approximation $\tilde{q}$ *to* q^**, such that* $\|q^* - \tilde{q}\|_\infty < \epsilon$*. The algorithm runs in time polynomial in* $|M|$ *and* $\log \frac{1}{\epsilon}$*, in the standard Turing model of computation.*

This follows from Theorem 3, using the fact that $\log(1/q^*_{\min})$ is polynomially bounded by Lemma 2, and the fact that the nonlinear depth of the MPS $x = P(x)$ for any p1CA is $f \leq 1$ ([10]). The detailed proof is in [21].

5.1 Application to ω-Regular Model Checking for p1CAs

Since computing termination probabilities of p1CAs (equivalently, the G-matrix of QBDs) plays such a central role in other analyses (see, e.g., [18, 3, 10, 4]), the P-time algorithm given in the previous section for computing termination probabilities of a p1CA (within arbitrary desired precision) directly facilitates P-time algorithms for various other important problems.

Here we highlight just one of these applications: a P-time algorithm *in the Turing model of computation* for model checking a p1CA with respect to any ω-regular property. An analogous result was established by Brazdil, Kiefer, and Kucera [4] in the unit-cost RAM model of computation.

Theorem 6. *Given a p1CA,* M*, with states labeled from an alphabet* Σ*, and with a specified initial control state* v*, and given an* ω*-regular property* $L(B) \subseteq \Sigma^\omega$*, which is specified by a non-deterministic Büchi automaton,* B*, let* $Pr_M(L(B))$ *denote the probability that a run of* M *starting at configuration* $(v, 0)$ *generates an* ω*-word in* $L(B)$*. There is an algorithm that, for any* $\epsilon > 0$*, computes an additive* ϵ*-approximation,* $\tilde{p} \geq 0$*, of* $Pr_M(L(B))$*, i.e., with* $|Pr_M(L(B)) - \tilde{p}| \leq \epsilon$*. The algorithm runs in time polynomial in* $|M|$*,* $\log \frac{1}{\epsilon}$*, and* $2^{|B|}$*, in the standard Turing model of computation.*

References

[1] Allender, E., Bürgisser, P., Kjeldgaard-Pedersen, J., Miltersen, P.B.: On the complexity of numerical analysis. SIAM J. Comput. 38(5), 1987–2006 (2009)

[2] Apostol, T.: Mathematical Analysis, 2nd edn. Addison-Wesley (1974)

[3] Bini, D., Latouche, G., Meini, B.: Numerical methods for Structured Markov Chains. Oxford University Press (2005)

[4] Brázdil, T., Kiefer, S., Kučera, A.: Efficient analysis of probabilistic programs with an unbounded counter. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 208–224. Springer, Heidelberg (2011)

[5] Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)

[6] Durbin, R., Eddy, S.R., Krogh, A., Mitchison, G.: Biological Sequence Analysis: Probabilistic models of Proteins and Nucleic Acids. Cambridge U. Press (1999)

[7] Esparza, J., Kiefer, S., Luttenberger, M.: Computing the least fixed point of positive polynomial systems. SIAM J. on Computing 39(6), 2282–2355 (2010)

[8] Esparza, J., Kučera, A., Mayr, R.: Model checking probabilistic pushdown automata. Logical Methods in Computer Science 2(1), 1–31 (2006)

[9] Etessami, K., Stewart, A., Yannakakis, M.: Polynomial-time algorithms for multitype branching processes and stochastic context-free grammars. In: Proc. 44th ACM Symposium on Theory of Computing, STOC (2012); Full version is available at ArXiv:1201.2374

[10] Etessami, K., Wojtczak, D., Yannakakis, M.: Quasi-birth-death processes, tree-like QBDs, probabilistic 1-counter automata, and pushdown systems. Performance Evaluation 67(9), 837–857 (2010)

[11] Etessami, K., Yannakakis, M.: Recursive markov chains, stochastic grammars, and monotone systems of nonlinear equations. Journal of the ACM 56(1) (2009)

[12] Etessami, K., Yannakakis, M.: Model checking of recursive probabilistic systems. ACM Trans. Comput. Log. 13(2), 12 (2012) (Conference versions in TACAS 2005 and QEST 2005)

[13] Hansen, K.A., Koucký, M., Lauritzen, N., Miltersen, P.B., Tsigaridas, E.P.: Exact algorithms for solving stochastic games: extended abstract. In: STOC, pp. 205–214 (2011); see full Arxiv version, arXiv:1202.3898 (2012)

[14] Harris, T.E.: The Theory of Branching Processes. Springer (1963)

[15] Horn, R.A., Johnson, C.R.: Matrix Analysis. Cambridge U. Press (1985)

[16] Isaacson, E., Keller, H.: Analysis of Numerical Methods. Wiley (1966)

[17] Lancaster, P., Tismenetsky, M.: The Theory of Matrices. Academic Press (1985)

[18] Latouche, G., Ramaswami, V.: Introduction to Matrix Analytic Methods in Stochastic Modeling. ASA-SIAM series on statistics and applied probability (1999)

[19] Nederhof, M.-J., Satta, G.: Computing partition functions of PCFGs. Research on Language and Computation 6(2), 139–162 (2008)

[20] Neuts, M.F.: Matrix-Geometric Solutions in Stochastic Models: an algorithmic approach. Johns Hopkins U. Press (1981)

[21] Stewart, A., Etessami, K., Yannakakis, M.: Upper bounds for Newton's method on monotone polynomial systems, and P-time model checking of probabilistic one-counter automata. arXiv:1302.3741 (2013) (full version of this paper)

[22] Wojtczak, D., Etessami, K.: PReMo: An analyzer for probabilistic recursive models. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 66–71. Springer, Heidelberg (2007)

Probabilistic Program Analysis with Martingales

Aleksandar Chakarov and Sriram Sankaranarayanan

University of Colorado, Boulder, CO.
firstname.lastname@colorado.edu

Abstract. We present techniques for the analysis of infinite state probabilistic programs to synthesize probabilistic invariants and prove almost-sure termination. Our analysis is based on the notion of (super) martingales from probability theory. First, we define the concept of (super) martingales for loops in probabilistic programs. Next, we present the use of concentration of measure inequalities to bound the values of martingales with high probability. This directly allows us to infer probabilistic bounds on assertions involving the program variables. Next, we present the notion of a super martingale ranking function (SMRF) to prove almost sure termination of probabilistic programs. Finally, we extend constraint-based techniques to synthesize martingales and super-martingale ranking functions for probabilistic programs. We present some applications of our approach to reason about invariance and termination of small but complex probabilistic programs.

1 Introduction

Probabilistic programs are obtained by enriching standard imperative programming languages with *random value generators* that yield sequences of (pseudo) random samples from some probability distribution. These programs commonly occur in a variety of situations including randomized algorithms [24,12], network protocols, probabilistic robotics, and monte-carlo simulations. The analysis of probabilistic programs is a rich area that has received a lot of attention in the past, yielding tools such as PRISM [18] that implement a wide variety of approaches ranging from symbolic [2] to statistical model checking [33,6].

Our goal in this paper is to investigate a deductive approach to infinite state probabilistic programs, exploring techniques for synthesizing *invariance* and *termination* proofs in a probabilistic setting. Our approach is an extension of McIver and Morgan's *quantitative invariants* [21,20]. Whereas the earlier approach is limited to *discrete probabilistic choices* (eg., Bernoulli trials), our extension handles probabilistic programs with integer and real-valued random variables according to a range of distributions including uniform, Gaussian and Poisson. We make use of the concepts of *martingales* and *super martingales* from probability theory to enable the synthesis of probabilistic invariants and almost sure termination proofs. A martingale expression for a program is one whose *expected value* after the $(n+1)^{th}$ loop iteration is equal to its sample value at the n^{th} iteration. We use Azuma-Hoeffding theorem to derive probabilistic bounds on the value of a martingale expression. Next, we present the concept of a *super martingale ranking function* (SMRF) to prove almost sure termination of loops that manipulate real-valued variables. Finally, we extend constraint-based program analysis techniques

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 511–526, 2013.

originally proposed by Colón et al. to yield techniques for inferring martingales and super-martingales using a template (super) martingale expression with unknown coefficients [8,7]. Our approach uses Farkas lemma to encode the conditions for being a (super) martingale as a system of linear inequality constraints over the unknown coefficients. However, our approach does not require non-linear constraint solving unlike earlier constraint-based approaches [7]. The constraint-based synthesis of linear (super) martingales yields *linear* inequality constraints. We present a preliminary evaluation of our approach over many small but complex benchmark examples using a prototype martingale generator.

The contributions of this paper are (a) we extend *quantitative invariants* approach of McIver and Morgan [20] to a wider class of probabilistic programs through martingale and super-martingale program expressions using concentration of measure inequalities (Azuma-Hoeffding theorem) to generate probabilistic assertions. (b) We define super martingale ranking functions (SMRFs) to prove almost sure termination of probabilistic programs. (c) We present constraint-based techniques to generate (super) martingale expressions. Some of the main limitations of the current work are: (a) currently, our approach applies to purely stochastic programs. Extensions to programs with demonic non-determinism will be considered in the future. (b) The martingale synthesis approach focuses on linear expressions and systems. Extensions to non-linear programs and expressions will be considered in the future. (c) Finally, our approach for proving almost sure termination is *sound* but incomplete over the reals. We identify some of the sources of incompleteness that arise especially for probabilistic programs.

Supplementary Materials: An extended version of this paper that includes many of the omitted technical details along with a prototype implementation of the techniques presented here will be made available on-line at our project page[1].

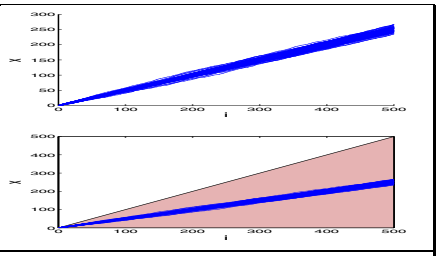
1.1 Motivating Examples

We motivate our approach on two simple examples of probabilistic programs.

Example 1. Figure 1 shows a program that accumulates the sum of samples drawn from a uniform random distribution between $[0, 1]$. Static analysis techniques can infer the invariant $0 \leq \mathrm{x} \leq 501 \ \wedge \ \mathrm{i} = \mathrm{N}$ at loop exit [10,22]. However, the value of x remains *tightly clustered* around 250, as seen in Fig. 1. Using the approaches in this paper, we can *statically* conclude the probabilistic assertion $\texttt{Pr}(x \in [200, 300]) \geq 0.84$. Unlike statistical model checkers, our symbolic technique provides guaranteed probability bounds as opposed to high confidence bounds [33]. For the example above, the behavior of x can also be deduced using the properties of sums of uniform random variables. However, our approach using martingale theory is quite general: the martingale and the inferred bounds hold even if the call to `unifRand` is substituted by (say) a truncated Gaussian distribution with mean $\frac{1}{2}$.

[1] `http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis`

```
1 real x = 0;
2 real N = 500;
3 for ( i=0; i < N; ++i )
4   x = x + unifRand(0,1);
5 // Prob(x \in [200,300]) ?
```




```
1 real h, t;
2 // h is hare and t is tortoise
3 h = 0; t = 30;
4 while ( h <= t ){
5   if (flip (0.5) )
6       h = h + unifRand(0,10);
7   t = t +1;
8 } // almost sure terminate?
```

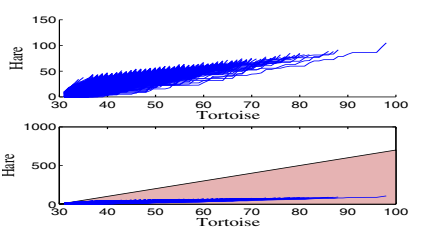


Fig. 1. (**Top, Left**) Program that sums up random variables. (**Top,Right**) sample paths of the program with value of i in x-axis and y-axis showing x. Loop invariants assuming (demonic) non-deterministic semantics for `unifRand` are shown on the right. (**Bottom,Left**) A simple probabilistic program with a loop, (**Bottom,Right**) Sample executions of the program with x axis representing t and y axis representing h. The plot on the bottom also contrasts the worst-case invariants inferred on h, t.

Example 2 (Almost Sure Termination). Consider the program shown in Figure 1. It shows a program that manipulates two real-valued variables h and t. Initially, the value of t is set to 30 and h to 0. The loop iterates as long as h $\leq$ t. Does the program terminate? In the worst case, the answer is no. It is possible that the coin flips avoid incrementing h or when it gets incremented, the uniform random value drawn lies in the interval $[0, 1]$. However, the techniques of this paper establish almost-sure termination using a *super martingale* expression t $-$ h. Such an expression behaves like a ranking function: It is initially positive and whenever its value is non-positive, the loop termination condition is achieved. Finally, for each iteration of the loop, the value of this expression decreases *in expectation* by at least 1.5. Therefore, the techniques presented in this paper infer the almost sure termination of this loop. Furthermore, we also conclude the martingale expression $2.5\text{t}-\text{h}$. We use Azuma-Hoeffding theorem to conclude that the value of this expression is tightly clustered around its initial value 75.

2 Probabilistic Transition Systems

In this section, we present a simple transition system model for probabilistic programs. Our model is inspired by the probabilistic guarded command language (PGCL) proposed by McIver et al. [21]. Unlike pGCL, our model allows non-discrete real-valued random variables with arbitrary distributions (Gaussian, Uniform, Exponential etc.). However, we do not allow (demonic) non-determinism. Let $X = \{x_1, \ldots, x_n\}$ be a set

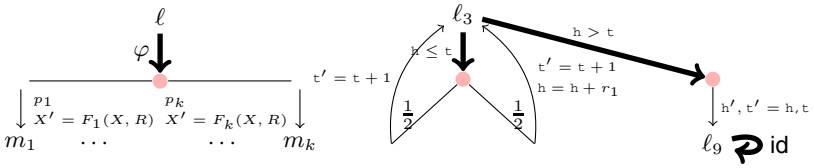


Fig. 2. (LEFT) Structure of a generic PTS transition with $k \geq 1$ forks. Each fork has a probability p_j, an update function F_j and a destination m_j. (MIDDLE) PTS for program in Figure 1 showing two transitions.

of real-valued *program variables* and $R = \{r_1, \ldots, r_m\}$ be a set of real-valued *random variables*. The random variables are assumed to have a joint distribution $\mathcal{D}$.

Definition 1 (Probabilistic Transition System). *A Probabilistic Transition System (PTS) Π is defined by a tuple $\langle X, R, L, \mathcal{T}, \ell_0, \boldsymbol{x}_0 \rangle$ such that*

1. *X, R represent the program and random variables, respectively.*
2. *L represents a finite set of* locations*. $\ell_0 \in L$ represents the* initial location*, and $\boldsymbol{x}_0$ represents the* initial values *for the program variables.*
3. *$\mathcal{T} = \{\tau_1, \ldots, \tau_p\}$ represents a finite set of* transitions*. Each transition $\tau_j \in \mathcal{T}$ is a tuple $\langle \ell, \varphi, f_1, \ldots, f_k \rangle$ consisting of (see Fig 2):*
   (a) Source location $\ell \in L$, and guard assertion φ over X,
   (b) Forks *$\{f_1, \ldots, f_k\}$, where each fork $f_j : (p_j, F_j, m_j)$ is defined by a fork probability $p_j \in (0, 1]$, a (continuous) update function $F_j(X, R)$ and a destination $m_j \in L$. The sum of the fork probabilities is $\sum_{j=1}^{k} p_j = 1$.*

No Demonic Restriction: We assume that all PTSs satisfy the *no demonic restriction*:

1. For each location ℓ, if τ_1 and τ_2 are any two different outgoing transitions at ℓ, then their guards φ_1 and φ_2 are *mutually exclusive*: $\varphi_1 \wedge \varphi_2 \equiv \mathit{false}$.
2. Let $\varphi_1, \ldots, \varphi_p$ be the guards of all the outgoing transitions at location ℓ. Their disjunction is *mutually exhaustive*: $\varphi_1 \vee \varphi_2 \vee \cdots \vee \varphi_p \equiv \mathit{true}$.

Mutual exhaustiveness is not strictly necessary. However, it simplifies our operational semantics considerably. The definition of a PTS seems quite involved at a first sight. We illustrate how probabilistic programs can be translated into PTS by translating the program in Example 2, as shown in Figure 2. The self loop on location ℓ_9 labeled id indicates a transition with guard true and a single fork with probability 1 that applies the identity function on the state variables. This transition is added to conform to the no demonic restriction. The semantics for a PTS Π are now described formally starting from the notion of a state and the semantics of each transition.

A *state* of the PTS is a tuple $(\ell, \boldsymbol{x})$ with location $\ell \in L$ and valuation $\boldsymbol{x}$ for the system variables X. We consider the effect of executing a single transition $\tau : \langle \ell, \varphi, f_1, \ldots, f_k \rangle$ on a state $s : (\ell, \boldsymbol{x})$. We assume that τ is *enabled* on s, i.e., $\boldsymbol{x} \models \varphi$. The result of executing τ on s is a *probability distribution* over the post states, obtained by carrying out the following steps:

1. Choose a fork f_j for $j \in [1,k]$ with probability p_j, and a vector of random variables $\boldsymbol{r} : (r_1, \ldots, r_m)$ drawn according to the joint distribution $\mathcal{D}$.
2. Update the states by computing the function $\boldsymbol{x}' = F_j(\boldsymbol{x}, \boldsymbol{r})$. The post-location is updated to $m_j \in L$.

Let POST-DISTRIB(s, τ) represent the distribution of the post states $(\ell', \boldsymbol{x}')$ starting from a fixed state $s : (\ell, \boldsymbol{x})$ for an enabled transition τ. A formal definition is provided in our extended version. Due to the no demonic restriction, exactly one transition τ is enabled for each state s. Therefore, the distribution POST-DISTRIB(s, τ) can simply be written as POST-DISTRIB(s) since τ is uniquely defined given s. Operationally, a PTS is a Markov chain, where each *sample execution* is a countable sequence of states.

Definition 2 (Sample Executions). *Let Π be a transition system. A sample execution σ of Π is a countably infinite sequence of states $\sigma : (\ell_0, \boldsymbol{x}_0) \xrightarrow{\tau_1} (\ell_1, \boldsymbol{x}_1) \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} (\ell_n, \boldsymbol{x}_n) \cdots$, such that (a) $(\ell_0, \boldsymbol{x}_0)$ is the unique initial state. (b) The state $s_j : (\ell_j, \boldsymbol{x}_j)$ for $j \geq 0$ satisfies the guard for the transition τ_{j+1}. Note that by the no demonic restriction, τ_{j+1} is uniquely defined for each s_j. (c) Each state $s_{j+1} : (\ell_{j+1}, \boldsymbol{x}_{j+1})$ is a sample from* POST-DISTRIB(s_j).

Figure 1 plots the sample paths for two probabilistic transition systems.

Almost-Sure Termination: Let Π be a PTS with a special *final location* ℓ_F. We assume that the only outgoing transition at the final location is the *identity* transition id, as defined earlier. A sample execution σ of Π *terminates* if it eventually reaches a state $(\ell_F, \boldsymbol{x})$, and thus, continues to cycle in the same state. Let $\pi : \ell_0 \xrightarrow{\tau_1} \ell_1 \xrightarrow{\tau_2} \cdots \ell_F$ be a syntactic path through the PTS. It can be shown that (a) for each finite syntactic path there is a well-defined probability $\mu(\pi) \in [0,1]$ that characterizes the probability that a sample path traverses the sequence of locations in π, and furthermore (b) the overall probability of termination can be obtained as the sum of the probabilities of all finite syntactic paths π_j that lead from ℓ_0 to ℓ_F.

Definition 3. *A PTS is said to terminate almost surely iff the sum of probabilities of the terminating syntactic paths is* 1.

A measure theoretic definition of almost sure termination is provided in our extended technical report.

Pre-expectation: We now define the useful concept of pre-expectation of an expression $\mathbf{e}$ over program variables across a transition τ, extending the definition for discrete probabilities from McIver & Morgan [20]. Let $s : (\ell, \boldsymbol{x})$ be a state and τ be the enabled transition on s. We define the pre-expectation $\mathbb{E}(\mathbf{e}'|s)$ as the expected value of the expression $\mathbf{e}'$ over the POST-DISTRIB(s), *as an expression involving the current state variables of the program*. Formally, let $\tau : (\ell, \varphi, f_1, \ldots, f_k)$ have $k \geq 1$ forks each of the form $f_j : (p_j, F_j, m_j)$. We define $\mathbb{E}_\tau(\mathbf{e}'|s) = \sum_{j=1}^{k} p_j \mathbb{E}_R(\text{PRE}(\mathbf{e}', F_j))$, where PRE$(\mathbf{e}', F_j)$ represents the expression $\mathbf{e}'[\boldsymbol{x}' \mapsto F_j(\boldsymbol{x}, \boldsymbol{r})]$ obtained by substituting the post-state variables $\boldsymbol{x}'$ by $F_j(\boldsymbol{x}, \boldsymbol{r})$, and $\mathbb{E}_R(g)$ represents the expectation of the expression g over the distribution $\mathcal{D}$. We clarify this further using an example.

Example 3. Going back to Example 2, we wish to compute the pre-expectation of the expression $5 \cdot \mathrm{t} - 2 \cdot \mathrm{h}$ across the transition $\tau_1 : (\ell_3, (\mathrm{h} \leq \mathrm{t}), f_1, f_2)$ with forks $f_1 : (\frac{1}{2}, F_1 : \lambda\,(\mathrm{h}, \mathrm{t}).\ (\mathrm{h}, \mathrm{t}+1), \ell_3)$ and $f_2 : (\frac{1}{2}, F_2 : \lambda(\mathrm{h}, \mathrm{t}).\ (\mathrm{h}+r_1, \mathrm{t}+1), \ell_3)$ (see Fig. 2). The precondition across F_1 yields $\textsc{pre}(5 \cdot \mathrm{t}' - 2 \cdot \mathrm{h}', F_1) = 5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5$. The precondition across F_2 yields $\textsc{pre}(5 \cdot \mathrm{t}' - 2 \cdot \mathrm{h}', F_2) = 5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5 - 2r_1$. The overall pre-expectation is given by

$$\mathbb{E}_\tau(5 \cdot \mathrm{t}' - 2 \cdot \mathrm{h}' | (\ell_3, \mathrm{h}, \mathrm{t})) = \begin{pmatrix} \frac{1}{2}(\mathbb{E}_{r_1}(5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5)) + & (* \leftarrow f_1 *) \\ \frac{1}{2}(\mathbb{E}_{r_1}(5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5 - 2r_1)) & (* \leftarrow f_2 *) \end{pmatrix}.$$

Simplifying, we obtain $\mathbb{E}(5 \cdot \mathrm{t}' - 2 \cdot \mathrm{h}' | (\ell_3, \mathrm{h}, \mathrm{t})) = 5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5 - \mathbb{E}_{r_1}(r_1) = 5 \cdot \mathrm{t} - 2 \cdot \mathrm{h} + 5 - 5 = 5 \cdot \mathrm{t} - 2 \cdot \mathrm{h}$. In other words, the expected value of the expression $5 \cdot \mathrm{t} - 2 \cdot \mathrm{h}$ in the post-state across transition τ is equal to its value in the current state. Such an expression is called a *martingale* [32].

The rest of this paper will expand on the significance of martingale expressions.

3 Martingales and Supermartingale Expressions

A discrete-time stochastic process $\{M_n\}$ is a countable sequence of random variables $M_0, M_1, M_2, \ldots$ where M_n is distributed based on the samples drawn from $M_0, \ldots, M_{n-1}$. By convention, M_n denotes the random variable and m_n its sample.

Definition 4 (Martingales and Super Martingales). *A process $\{M_n\}$ is a* martingale *iff for each $n > 0$, $\mathbb{E}(M_n | m_{n-1}, \ldots, m_0) = m_{n-1}$. In other words, at each step the expected value at the next step is equal to the current value. Likewise $\{M_n\}$ is a* super-martingale *iff for each $n > 0$, $\mathbb{E}(M_n | m_{n-1}, \ldots, m_0) \leq m_{n-1}$.*

If $\{M_n\}$ is a martingale then it is also a super-martingale. Furthermore, the process $\{-M_n\}$ obtained by negating M_n is also a super-martingale. The study of martingales is fundamental in probability theory [32] with numerous applications to the analysis of randomized algorithms [12].

Martingale Expressions: Let Π be a PTS with locations L, variables X and random variables R. We assume, for convenience, that all variables in X and random values are real-valued. We define martingale expressions for a given PTS Π. Next, we explore properties of martingales and super martingales, linking them to those of the PTS.

Definition 5 (Martingale Expressions). *An expression $\mathbf{e}[X]$ over program variables X is a* martingale *for the PTS Π iff for every transition $\tau : (\ell, \varphi, f_1, \ldots, f_k)$ in Π and for every state $s : (\ell, \boldsymbol{x})$ for which τ is enabled, the pre-expectation of $\mathbf{e}$ equals its current state value: $\forall\, \boldsymbol{x}.\ \varphi[\boldsymbol{x}] \Rightarrow \mathbb{E}_\tau(\mathbf{e}' | \ell, \boldsymbol{x}) = \mathbf{e}$. Likewise, an expression is a* super-martingale *iff for each transition τ, $\forall\, \boldsymbol{x}.\ \varphi[\boldsymbol{x}] \Rightarrow \mathbb{E}_\tau(\mathbf{e}' | \ell, \boldsymbol{x}) \leq \mathbf{e}$.*

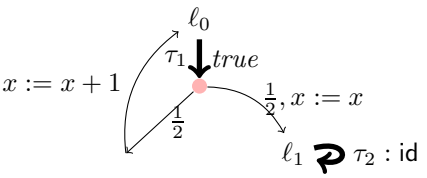
In other words, the stochastic process $\{\mathbf{e}_n\}$ obtained by evaluating the expression $\mathbf{e}$ on a sample execution of Π must be a (super) martingale. Note that in the terminology of McIver & Morgan, martingale expressions correspond to *exact invariants* [21].

Example 4. We noted that for the expression $e : 5\cdot \mathrm{t} - 2\cdot \mathrm{h}$, and for any τ_1 enabled state $(\ell_3, \mathrm{h}, \mathrm{t})$, we have that $\mathbb{E}(5\cdot \mathrm{t}' - 2\cdot \mathrm{h}' | \mathrm{h}, \mathrm{t}) = 5\cdot \mathrm{t} - 2\cdot \mathrm{h}$. The remaining transitions in the PTS (Cf.Fig. 2) also preserve the expression e. Therefore, e is a martingale. Likewise, we can show that the expression $f : -\mathrm{h}$ is a super-martingale.

Often, it is not possible to obtain a single expression that is a martingale for the program as a whole. However, it is natural to obtain a more complex function of the state that maps to different program expressions at different locations.

Example 5. Consider a simple program and its corresponding PTS, as shown below. The program increments a variable x as long as a coin flip turns heads.

```
1  int x := 0;
2  while (flip (0.5))
3      x ++;
4  // end
```



We can show that no linear expression over x can be a martingale. Any such expression must be of the form $\mathbf{e} : cx$ (we do not need a constant term in our analysis) for some coefficient c. We note that the pre-expectation w.r.t τ_1 is $\mathbb{E}_{\tau_1}(cx'|x) = \frac{1}{2}(c(x+1)) + \frac{1}{2}(cx) = cx + \frac{c}{2}$. This yields the constraint $c = 0$ if the expression is to be a martingale. However, consider a *flow-sensitive* map of the state (ℓ, x) given by $f(\ell, x) = \begin{cases} x & \text{if } \ell = \ell_0 \\ x - 1 & \text{if } \ell = \ell_1 \end{cases}$. We can conclude that $f(\ell, x)$ is a martingale, since its pre-expectation for any transition equals its current value.

Often, labeling different locations in the program with different martingale expressions is quite advantageous.

Definition 6 (Flow-Sensitive Expression Maps). *A flow-sensitive* expression map η *maps each location $\ell \in L$ to a (polynomial) expression $\eta(\ell)$ over X.*

An expression map η is a function that maps a state $s : (\ell, \boldsymbol{x})$ to a real-value by computing $\eta(\ell)[\boldsymbol{x}]$. Let η be an expression map and $\tau : (\ell, \varphi, f_1, \ldots, f_k)$ be a transition with forks $f_1, \ldots, f_k$, where $f_j : (p_j, F_j, m_j)$. The *pre-expectation* of η w.r.t. τ is given by $\mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) = \sum_{j=1}^{k} p_j \mathbb{E}_R(\mathrm{PRE}(\eta(m_j), F_j))$.

Example 6. For the program and map η in Ex. 5, we compute the pre-expectation of η w.r.t $\tau_1 : (\ell_0, true, f_1, f_2)$ where $f_1 : (\frac{1}{2}, \lambda x.x + 1, \ell_0)$ and $f_2 : (\frac{1}{2}, \mathsf{id}, \ell_1)$.

$$\mathbb{E}_{\tau_1}(\eta'|\ell_0, x) = \frac{1}{2}(\underbrace{x+1}_{\mathrm{PRE}(\eta(\ell_0), \lambda x.x+1)}) + \frac{1}{2}(\underbrace{x-1}_{\mathrm{PRE}(\eta(\ell_1), \lambda x.x)}) = x(= \eta(\ell_0)) \,.$$

Definition 7 (Martingales and Super Martingale Expression Maps). *An expression map η is a martingale for a PTS Π iff for every transition $\tau : (\ell, \varphi, f_1, \ldots, f_k)$, we have $\forall\, \boldsymbol{x}.\ \varphi[\boldsymbol{x}] \Rightarrow \mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) = \eta(\ell)[\boldsymbol{x}]$.*

Likewise, the map is a super-martingale iff for every transition τ, $\forall\, \boldsymbol{x}.\ \varphi[\boldsymbol{x}] \Rightarrow \mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) \leq \eta(\ell)[\boldsymbol{x}]$.

Example 7. The map η in Example 6 is a martingale under the condition for transition τ_2. The only other transition is trivial.

3.1 From Martingales to Probabilistic Assertions

We now present some of the key properties of martingales that can be used to make a link from (super) martingale expression maps to probabilistic assertions.

Theorem 1 (Azuma-Hoeffding Theorem). *Let $\{M_n\}$ be a super martingale such that $|m_n - m_{n-1}| < c$ over all sample paths for constant c. Then for all $n \in \mathbb{N}$ and $t \in \mathbb{R}$ such that $t \geq 0$, it follows that $\texttt{Pr}(M_n - M_0 \geq t) \leq \exp\left(\frac{-t^2}{2nc^2}\right)$. Moreover, if $\{M_n\}$ is a martingale the symmetric bound holds as well: $\texttt{Pr}(M_n - M_0 \leq -t) \leq \exp\left(\frac{-t^2}{2nc^2}\right)$. Combining both bounds, we conclude that for a martingale $\{M_n\}$ we obtain $\texttt{Pr}(|M_n - M_0| \geq t) \leq 2\exp\left(\frac{-t^2}{2nc^2}\right)$.*

Azuma-Hoeffding bound is a *concentration of measure inequality*. For a martingale, it bounds the probability of a large deviation on either side of its starting value. For a super-martingale, the inequality bounds the probability of a large deviation *above* the starting value. Both bounds are useful in proving probabilistic assertions.

We note the condition of *bounded change* on the martingale. This has to be established on the side for each transition in the program, and the bound c calculated (using optimization techniques) before the inequality can be applied.

Example 8. The Azuma-Hoeffding bound applies to the program in Ex. 1 (Section 1). We now observe that the expression $2\texttt{x} - \texttt{i}$ is a martingale of the loop. Its change at any step is bounded by ± 1. Further, the initial value of the expression is 0. Therefore, choosing $t = 50$, we conclude that after $N = 500$ steps, $\texttt{Pr}(|(2\texttt{x} - \texttt{i})_{500} - (2\texttt{x} - \texttt{i})_0| \geq 50) \leq 2\exp\left(-\frac{2500}{2\times 500\times 1}\right) \leq 0.16$. We note that $(2\texttt{x} - \texttt{i})_0 = 0$ and $\texttt{i}_{500} = 500$. Simplifying, with probability at least 0.84, we conclude that $x \in [200, 300]$ after 500 steps.

Since the bounds depend on the number of steps n taken from the start, they are easiest to apply when n is fixed or bounded in an interval. Another common idea is to infer bounds $|M_n - M_0| \geq a\sqrt{n}$ for constant $a > 0$ and $n \geq 0$.

Example 9. Continuing with Ex. 8, for $n > 0$, we conclude the bounds $\texttt{Pr}(|(2\texttt{x} - \texttt{i})_n - (2\texttt{x} - \texttt{i})_0| \geq a\sqrt{n}) \leq 2\exp(-\frac{a^2 n}{2n}) \leq 2\exp(-\frac{a^2}{2})$. For $a = 3$, the upper bound is 0.0223.

4 Almost Sure Termination

In this section, we provide a technique for proving that a PTS Π with a final location ℓ_F is almost surely terminating (see definition of almost sure termination in Section 2).

Definition 8 (Ranking Super Martingale). *A super martingale (s.m.) $\{M_n\}$ is* ranking *iff it has the following properties:*

1. *There exists* $\epsilon > 0$ *such that for all sample paths,* $\mathbb{E}(M_{n+1}|m_n) \leq m_n - \epsilon$.
2. *For all* $n \geq 0$, $M_n \geq -K$ *for some constant* $K > 0$.

Let $\{M_n\}$ be a ranking s.m. with positive initial condition $m_0 > 0$. A sample path eventually *becomes negative* if $m_n \leq 0$ for some $n \geq 1$.

Theorem 2. *A ranking super martingale with a positive initial condition almost surely becomes negative.*

Proof. The *stopping time* for a sample path is defined as $t = \inf_{n \geq 0} m_n \leq 0$. We use T as a random variable for the stopping time. The *stopped process* is denoted $M_{\min(n,T)}$ (or M_n^T) has sample paths $m_0, \ldots, m_t, m_t, m_t, \ldots$. Note that $M_{\min(n,T)} \geq -K$ over all sample paths.

Next, we define a process $Y_n = M_{\min(n,T)} + \epsilon \min(n, T)$. In other words, for each sample path $y_n = m_n + \epsilon n$ if $n \leq t$, and $y_n = m_t + \epsilon t$ if $n > t$. Note that given a sample path prefix $m_0, \ldots, m_n$ of M_n^T we can compute y_n. Therefore, Y_n is *adapted* to $M_{\min(n,T)}$. Likewise, given $y_0, \ldots, y_n$ we can compute $m_{\min(n,t)}$. Therefore, sample paths of Y_n are one-to-one correspondent with those of M_n^T.

Lemma 1. $\{Y_n\}$ *is a super martingale (relative to* M_n^T*) and* $Y_n \geq -K$.

Proof. $Y_n \geq -K$ for all n follows from the fact that $M_n^T \geq -K$ for all n. Next, we show that Y_n is a s.m. For any sample path, $\mathbb{E}(Y_{n+1}|y_n, m_n) = \mathbb{E}(M_{n+1}|y_n, m_n) + \min(n+1, t)\epsilon$. We split two cases (a) $n + 1 \leq t$ or (b) $n + 1 > t$.

Case (a): $\mathbb{E}(Y_{n+1}|y_n, m_n) = \mathbb{E}(M_{n+1}|m_n) + (n+1)\epsilon \leq m_n - \epsilon + (n+1)\epsilon \leq y_n$.

Case (b): $y_n = m_t + t\epsilon$. We have $\mathbb{E}(Y_{n+1}|m_n, y_n) = m_t + t\epsilon = y_n$.

In either case, we conclude $\mathbb{E}(Y_{n+1}|m_n, y_n) \leq y_n$.

We note the well-known s.m. convergence theorem.

Theorem 3 (Super Martingale Convergence Theorem). *A lower-bounded super martingale converges (samplewise) almost surely.*

Therefore, with probability 1, a sample path $y_0, \ldots, y_n, \ldots$ converges to a value $\tilde{y}$.

Lemma 2. *For any convergent sample path* $y_0, \ldots, y_n, \ldots$*, the corresponding* $\{M_n\}$ *sample path* $m_0, \ldots, m_n, \ldots$ *eventually becomes negative.*

Proof. Convergence of y_n to $\tilde{y}$ implies for any $\alpha > 0$, there exists N such that $\forall n \geq N, |y_n - \tilde{y}| \leq \alpha$. For contradiction, assume the $\{M_n\}$ sample path has stopping time $t = \infty$. Therefore, $m_n = y_n - n\epsilon$ for all $n \geq 0$. Choosing $\alpha = \epsilon$, for any $n > N$, $m_n \leq \tilde{y} + \alpha - n\epsilon \leq \tilde{y} - (n-1)\epsilon$. Therefore, for $n > 1 + \frac{\tilde{y}}{\epsilon}$, we conclude that $m_n \leq 0$. This contradicts our assumption that $t = \infty$.

To complete the proof, we observe that (a) a sample path $y_0, \ldots, y_n, \ldots$ converges almost surely since $\{Y_n\}$ is a lower bounded s.m.; (b) for each convergent sample path the corresponding (unique) path $m_0, \ldots, m_n, \ldots$ becomes negative; and therefore (c) any $\{M_n\}$ sample path becomes negative almost surely.

Definition 9 (Super Martingale Ranking Function). *A super martingale ranking function (SMRF)* η *is a s.m. expression map that satisfies the following:*

- $\eta(\ell) \geq 0$ *for all* $\ell \neq \ell_F$*, and* $\eta(\ell_F) \in [-K, 0)$ *for some lower bound* K.
- *There exists a constant* $\epsilon > 0$ *s.t. for each transition* τ *(other than the self-loop* id *around* ℓ_F*) with guard* φ*,* $(\forall\, \boldsymbol{x})\ \varphi[\boldsymbol{x}] \Rightarrow \mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) \leq \eta(\ell)[\boldsymbol{x}] - \epsilon$.

The SMRF definition above is a generalization of similar rules over discrete spaces, including the probabilistic variant rule [20] and Foster's theorem [14,4].

Theorem 4. *If a PTS* Π *has a super martingale ranking function* η *then every sample execution of* Π *terminates* almost surely.

For any sample execution of Π, we define the process $\{M_n\}$ where $m_n = \eta(s_n)$. It follows that $\{M_n\}$ is a ranking super martingale. The rest follows from Theorem 2

Example 10. For the PTS in Example 1, a.s. termination is established by the SMRF $\eta(\ell_0) : N - i$, and $\eta(\ell_1) : -1$. Consider the PTS in Example 2 and Figure 2, the SMRF $\eta(\ell_3) : \mathrm{t} - \mathrm{h} + 9$ and $\eta(\ell_9) : \mathrm{t} - \mathrm{h}$ proves a.s. termination. The PTS in Ex. 5 has a SMRF $\eta(\ell_2) : 1$ and $\eta(\ell_4) : -1$.

Unlike standard ranking functions, we do not obtain completeness. Consider a purely symmetric random walk:

```
1  int x := 10; while (x >= 0) { if (flip(0.5)) x++; else x --; }
```

We can show using recurrence properties of symmetric random walks, that the program above terminates almost surely. Yet, no SMRF can be found since the martingale x does not show adequate decrease. However, if the flip probability is changed to $0.5 - \delta$, then the variable x is a SMRF for the program.

5 Discovering (Super) Martingales

Next, we turn our attention to the discovery of (super) martingale expression maps and super martingale ranking functions (SMRF). Our approach builds upon previous work by Colón et al. for constraint-based invariant and ranking function discovery for standard non-deterministic transition systems [7,8]. We restrict our approach to *affine PTS* wherein each transition $\tau : (\ell, \varphi, f_1, \ldots, f_k)$, the guard φ is *polyhedral* (conjunctions of linear inequalities) and the update function F_i for each f_i is affine, $F_i(X, R) : A_i\boldsymbol{x} + B_i\boldsymbol{r} + \boldsymbol{a_i}$.

A *template expression* is a bilinear form $d + \sum_{i=1}^{n} c_i x_i$ with unknowns $c_1, \ldots, c_n, d$. We may also consider a template expression map η that maps each location ℓ_j to a template expression $\eta(\ell_j) : d_j + \sum_{i=1}^{n} c_{ji} x_i$. We collectively represent the unknown coefficients as a vector $\boldsymbol{c}$. We encode the conditions for a template expression (map) corresponding to our objective: (super) martingales or super martingale ranking functions. Solving the resulting constraints directly yields (super) martingales.

Example 11. Consider the PTS in example 2 (see Fig. 2). We wish to discover a s.m. using the template $c_1\mathrm{h} + c_2\mathrm{t}$ at locations ℓ_3, ℓ_9.

Encoding Super Martingales: We discuss how the conditions on the pre-expectations of s.m. can be encoded using Farkas Lemma. Let $\tau : (\ell, \varphi, f_1, \ldots, f_k)$ be a transition with k forks. Let η be a template expression map. We wish to enforce that η is a s.m. $(\forall\ \boldsymbol{x})\ (\varphi[\boldsymbol{x}]) \Rightarrow \mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) \leq \eta(\ell)[\boldsymbol{x}]$. Recall that $\mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) = \sum_{j=1}^{k} p_j \mathbb{E}_R(\text{PRE}(\eta(m_j), F_j))$. Each $\text{PRE}(\eta(m_j), F_j)$ can be expressed as $\text{PRE}(\eta(m_j), F_j) = \boldsymbol{c}^T A\boldsymbol{x} + \boldsymbol{c}^T B\boldsymbol{r} + \boldsymbol{c}^T \boldsymbol{a}$.

Example 12. Returning back to Ex. 11, we wish to encode pre-expectation condition for the transition $\tau : (\ell_3, (\mathrm{h} \leq \mathrm{t}), f_1, f_2)$, where $f_1 : (\frac{1}{2}, (\lambda(\mathrm{h}, \mathrm{t}).\ \mathrm{h}, \mathrm{t}+1), \ell_3)$ and $f_2 : (\frac{1}{2}, (\lambda(\mathrm{h}, \mathrm{t}).\ \mathrm{h}+r_1, \mathrm{t}+1), \ell_3)$. We encode the pre-expectation condition for τ:

$$(\forall \mathrm{h}, \mathrm{t})\ (\mathrm{h} \leq \mathrm{t}) \Rightarrow \begin{bmatrix} \frac{1}{2}\mathbb{E}_{r_1}(c_1\mathrm{h} + c_2(\mathrm{t}+1)) + \\ \frac{1}{2}\mathbb{E}_{r_1}(c_1(\mathrm{h}+r_1) + c_2(\mathrm{t}+1)) \end{bmatrix} \leq c_1\mathrm{h} + c_2\mathrm{t}\,.$$

We note that $\mathbb{E}_{r_1}(r_1) = 5$ (See Figure 1). By linearity of expectation, we obtain

$$(\forall \mathrm{h}, \mathrm{t})\ (\mathrm{h} \leq \mathrm{t}) \Rightarrow c_1\mathrm{h} + c_2\mathrm{t} + c_2 + \frac{1}{2}c_1\mathbb{E}_{r_1}(r_1) \leq c_1\mathrm{h} + c_2\mathrm{t}\,.$$

Simplifying, we obtain $(\forall \mathrm{h}, \mathrm{t})\ (\mathrm{h} \leq \mathrm{t}) \Rightarrow c_2 + \frac{5}{2}c_1 \leq 0$. Here, the RHS is independent of the variables h, t. Therefore, we obtain $c_2 + \frac{5}{2}c_1 \leq 0$.

Let $\boldsymbol{\mu}$ represent the vector of mean values where $\boldsymbol{\mu}_j = \mathbb{E}_R(r_j)$. Therefore,

$$\mathbb{E}_R(\text{PRE}(\eta(m_j), F_j)) = \boldsymbol{c}^T A\boldsymbol{x} + \boldsymbol{c}^T B\boldsymbol{\mu} + \boldsymbol{a}^T \boldsymbol{c}\,.$$

To encode the s.m. property for τ, we use Farkas Lemma to encode the implication

$$(\forall\ \boldsymbol{x})\ (\varphi[\boldsymbol{x}]) \Rightarrow \underbrace{\mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x})}_{\text{template expression}} \leq \underbrace{\eta(\ell)[\boldsymbol{x}]}_{\text{template expression}} \tag{1}$$

Let φ be satisfiable and represented in the constraint form as $A\boldsymbol{x} \leq \boldsymbol{b}$.

Theorem 5 (Farkas Lemma). *The linear constraint $A\boldsymbol{x} \leq \boldsymbol{b} \Rightarrow \boldsymbol{c}^T\boldsymbol{x} \leq d$ is* valid *iff its* alternative *is satisfiable $A^T\boldsymbol{\lambda} = \boldsymbol{c} \wedge \boldsymbol{b}^T\boldsymbol{\lambda} \geq d \wedge \boldsymbol{\lambda} \geq 0$.*

Encoding the entailment in Eq. (1) using Farkas' Lemma ensures that the resulting constraints are linear inequalities.

Example 13. Continuing with Ex. 12, the transition id yields the constraint $true$. Therefore, the only constraint is $c_2 + \frac{5}{2}c_1 \leq 0$. Solving, we obtain the line $(c_1 : 1, c_2 : \frac{-5}{2})$, yielding the martingale $\mathrm{h} - \frac{5}{2}\mathrm{t}$, while the ray $(c_1 : -1, c_2 : 0)$ yields the s.m. $-\mathrm{h}$. Other s.m. such as $\mathrm{t} - \mathrm{h}$ are obtained as linear combinations.

Finding Super Martingale Ranking Functions: The process of discovering SMRFs is quite similar, but requires extra constraints. An abstract interpretation pass can be used to yield helpful invariants by treating the random variables and forks as non-deterministic choices. Let $I(\ell)$ be a polyhedral invariant inferred at the location ℓ.

```
real x,y, estX, estY := 0,0,0,0
real dx, dy, dxc, dyc := 0,0,0,0
int i, N := 0,500
for i = 0 to N {
  cmd := choice(N:0.1,S:0.1,
    E:0.1,W:0.1,NE:0.1,SE:0.1,
    NW:0.1,SW:0.1,Stay:0.2)
  switch (cmd) {
    N: dxc,dyc := 0, rand(1,2)
    S: dxc, dyc := 0, -rand(1,2)
    Stay: dxc,dyc := 0,0
    E: dxc,dyc := rand(1,2), 0
    ...
  }
  dx:= dxc+rand(-.05,.05)
  dy:= dyc+rand(-.05,.05)
  x := x + dx
  y := y + dy
  estX := estX + dxc
  estY := estY + dyc }
```

```
int i := 0;
real money := 10,  bet
while (money >= 10 ) {
  bet := rand(5,10)
  money := money - bet
  if (flip(36/37)) // bank lost
  if flip(1/3)  //  col. 1
     if flip(1/2)
        money := money + 1.6*bet // Red
     else  money := money + 1.2*bet // Black
   elseif flip(1/2) //  col. 2
      if flip(1/3)
         money := money + 1.6*bet; // Red
      else  money := money + 1.2*bet // Black
   else  //  col. 3
      if flip(2/3)
     money := money + 0.4*bet // Red
  i := i + 1 }
```

Fig. 3. (**Left**) Probabilistic program model for dead reckoning and (**Right**) Modeling a betting strategy for Roulette

1. To encode the non-negativity, we use the invariants at each location $\ell \neq \ell_F$, $I(\ell) \models \eta(\ell) \geq 0$. For location ℓ_F, we encode $I(\ell_F) \models -K \leq \eta(\ell_F) < 0$. The latter condition requires the Motzkin's transposition theorem, a generalization of Farkas' lemma that deals with strict inequalities [17]. Here K is treated as an unknown constant, whose value is also inferred as part of the process.
2. The adequate decrease condition is almost identical to that for s.m. However, we introduce an unknown $\epsilon > 0$ and require that $\mathbb{E}_\tau(\eta'|\ell, \boldsymbol{x}) \leq \eta(\ell) - \epsilon$.

Example 14. Revisiting Ex. 12, we perform an abstract interpretation to obtain the facts $I(\ell_3) : 0 \leq \mathrm{h} \leq \mathrm{t} + 9 \ \wedge \ \mathrm{h} \leq 9\mathrm{t} - 270 \wedge \ \mathrm{t} \geq 30$ and $I(\ell_9) : \mathrm{h} > \mathrm{t} \ \wedge \ \mathrm{h} \leq \mathrm{t} + 9$. We use the template $\eta(\ell_3) : c_{3,1}\mathrm{h} + c_{3,2}\mathrm{t} + d_3$ and $\eta(\ell_9) : c_{9,1}\mathrm{h} + c_{9,2}\mathrm{t} + d_9$. We obtain the result $c_{3,1} = c_{9,1} = -1, c_{3,2} = c_{9,2} = 1$ and $d_3 = 10, d_9 = 0$, with $\epsilon = \frac{3}{2}$ and $K = -9$. This yields the SMRF $\eta(\ell_3) : \mathrm{t} - \mathrm{h} + 9, \ \eta(\ell_9) : \mathrm{t} - \mathrm{h}$.

6 Evaluation

We have implemented the ideas presented thus far using a constraint generation framework that reads in the description of a PTS and generates constraints for super-martingale expression maps. Our tool uses the Parma Polyhedra Library [1] to generate all possible solutions to these constraints in terms of martingale and super-martingale expressions. Currently, our implementation does not directly communicate with an abstract interpreter for deriving useful invariants. In some of the examples presented in this section, such invariants are computed using a numerical domain polyhedral abstract interpreter and added manually to the PTS description.

Robot Dead Reckoning: Dead reckoning is an approach for position estimation starting from a known *fix* at some time $t = 0$. Figure 3 (**left**) shows a model for robot

Table 1. Results on a set of benchmark programs. **#M:** # of non-trivial martingales discovered and # **S.M.** # of non-trivial super martingales. All timings are under 0.1 seconds on Macbook air laptop with 8 GB RAM, running MAC OSX 10.8.3.

ID	Description	$\|X\|$	$\|R\|$	$\|L\|$	$\|T\|$	#M	#S.M.
ROULETTE	betting strategy for roulette	3	1	1	1	1	1
TRACK	Target tracking with feedback	3	5	3	9	1	3
2DWALK	Random walk on $\mathbb{R}^2$	4	1	1	4	3	1
COUPON5	coupon collectors with $n = 5$ coupons	2	0	5	4	4	8
FAIRBIASCOIN	simulating a fair coin by biased coin	3	0	2	3	0	2
QUEUE	queue with random arrivals/service	3	0	1	2	1	2
CART	steering a cart on a rough surface	5	4	6	12	2	4
INVPEND	discrete inverted pendulum under stochastic disturbance	5	6	3	3	0	0
PACK	packing variable weight objects in cartons	6	2	3	5	3	4
CONVOY2	leader following over a convoy of cars	6	1	2	4	1	0
DRECKON	dead reckoning model	10	4	3	3	4	1

navigation that involves estimating the actual position (x,y) of the robot as it is commanded to make various moves. Each step involves a choice of direction chosen from the set of compass directions $\{N, W, E, S, NE, NW, SW, SE\}$ each with probability 0.1 or a "Stay" command with probability 0.2. The variables dxc,dyc capture the commanded direction, whereas the actual directions are slightly off by a random value. Our goal is to estimate how the position x,y deviates from the actual position estX,estY.

Our analysis shows that the expressions $\mathrm{x} - \mathrm{estX}$ and $\mathrm{y} - \mathrm{estY}$ are martingales at the loop head. The absolute change in these martingales are bounded by 0.05. Given the initial difference of 0 between the values, we infer using Azuma-Hoeffding theorem that $\texttt{Pr}(|\mathrm{x} - \mathrm{estX}| \geq 3) \leq 1.5 \times 10^{-3}$. In contrast, a worst-case analysis concludes that $|\mathrm{x} - \mathrm{estX}| \leq 0.05 * 500 \leq 25$. The analysis for $\mathrm{y} - \mathrm{estY}$ yields identical results.

Roulette: For our next example, we analyze a betting strategy for a game of Roulette. The game involves betting money on a *color* (red or black) and a column (1,2 or 3). At each step, the player chooses an amount to bet randomly between 5 and 10 dollars. We skip a detailed description of the betting strategy and simply model the effect of the strategy as a probabilistic program, as shown in Figure 3 (**right**). The model captures the various outcome combinations ($\{Bank\} \uplus \{\text{Red, Black}\} \times \{1, 2, 3\}$), including the one where the bank wins outright with probability $\frac{1}{37}$. Our analysis discovers the martingale expression $15 \times \mathrm{i} - 74 \times \text{money}$ which can be used to bound the probability of the money exceeding a certain quantity after n rounds. We generate the SMRF: $-$money. Thus, the program terminates almost surely in the gambler's *ruin*.

Table 1 shows an evaluation of our approach over a set of linear PTS benchmarks. A description of the benchmarks and the inferred properties are provided in our extended technical report available on-line[2].

[2] `http://systems.cs.colorado.edu/research/cyberphysical/probabilistic-program-analysis`

7 Related Work

Probabilistic programs can be quite hard to reason about. A large volume of related work has addressed techniques for formally specifying and verifying properties of probabilistic systems. Statistical approaches rely on simulations [33,6,19], providing high confidence guarantees. On the other hand, symbolic techniques including BDD-based approaches [2], probabilistic CEGAR [16], deductive approaches [21] and abstract interpretation techniques [23,11,3] attempt to establish guaranteed probability bounds on temporal properties of programs.

Martingale theory has been employed to establish guarantees for randomized algorithms [24,12]. In particular, the *method of bounded differences* is a popular approach that establishes a martingale and uses Azuma's inequality to place bounds on its values. The contribution of this paper lies in a partial mechanization of the process of discovering martingales and the termination analysis of programs using super-martingale ranking functions.

Our work is closely related to the *quantitative invariants* proposed by McIver and Morgan, and summarized in their monograph [20]. Informally, quantitative invariants involve program expressions whose pre-expectations are at least their current value. These are used to establish pre-/post-annotations for programs, and in some cases they lead to an *almost sure termination* proof principle. In the framework of this paper, a quantitative invariant roughly corresponds to the negation of a super-martingale (also known as a sub-martingale). There are many differences between McIver & Morgan's approach and that of this paper. Chiefly, our approach considers real-/integer-valued random variables with a large variety of probability distributions, whereas the probabilistic distributions studied by McIver & Morgan are restricted to discrete distributions over a finite set of choices. The use of concentration of measure inequalities and the presentation of almost sure termination proofs using martingale theory are unique to this work. On the other hand, our work does not consider (demonic) non-determinism. Furthermore, we do not integrate martingales and super-martingales into a deductive proof system for proving properties of expectations. Part of the reason for this lies in the difficulty of establishing that expectations of program variables are well-defined for a given program. Many simple examples fail to yield well-defined expectations. We plan to study these issues further and achieve a more complete deductive verification framework as part of our future work.

The constraint-based analysis of (non-probabilistic) programs has been studied for invariance and termination proofs by many authors, including Colón et al. [8,7], Bradley et al. [5], Cousot [9], Podelski et al. [28] and Gulwani et al. [15]. Extensions to polynomial invariants were considered by Sankaranarayanan et al. [31], Carbonell et al. [30], Müller-Olm et al. [25] and Platzer et al. [26]. Recent work of Katoen et al. uses a constraint-based invariant synthesis method to derive quantitative invariants, generalizing earlier approaches [17]. Our work in this paper was inspired, in part, by this generalization. Katoen et al. derive quantitative invariants that involve a combination of characteristic functions over linear assertions and program expressions. As a result, their approach yields nonlinear (bilinear) constraints of the same form as those obtained by earlier work by Colón et al. [7]. Our approach focuses on linear (super) martingales and therefore, we obtain linear inequality constraints.

The almost sure termination of probabilistic programs was studied for finite state and "weakly" infinite programs by Esparza et al. using *patterns* [13]. Their approach attempts to find the existence of a sequence of discrete probabilistic choices that will lead to termination from any reachable state. As such, finite length patterns do not exist for general infinite state systems studied here. Other approaches to almost sure termination including that of Morgan [20] and Bournez et al. [4] use the *probabilistic variant rule*, or equivalently *Foster's theorem*, a well-known result in (discrete) Markov chain theory [14]. In fact, these principles turn out to be specializations of the SMRF principle presented in this paper.

(Super) martingale theory is widely used in stochastic calculus to reason about continuous time systems. Recently, Platzer considers an extension of differential logic to stochastic hybrid systems [27]. Martingale theory is used as a basis for proving properties in this setting. However, Platzer's work does not deal directly with martingale expressions over state variables or the generation of such expressions. The techniques presented in this paper are dependent on discrete time martingale theory. We plan to extend our use of martingale theory to reason about stochastic hybrid systems. Another future direction will consider the use of (super) martingales to infer relational abstractions of probabilistic systems [29].

Acknowledgments. Thanks to Prof. Manuel Lladser for useful pointers to martingale theory and the anonymous reviewers for detailed comments. This work was supported by the US National Science Foundation (NSF) under award numbers CNS-0953941 and CNS-1016994.

References

1. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1-2), 3–21 (2008)
2. Baier, C., Clarke, E., Hartonas-Garmhausen, V., Kwiatkowska, M., Ryan, M.: Symbolic model checking for probabilistic processes. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 430–440. Springer, Heidelberg (1997)
3. Bouissou, O., Goubault, E., Goubault-Larrecq, J., Putot, S.: A generalization of p-boxes to affine arithmetic. Computing (2012)
4. Bournez, O., Garnier, F.: Proving positive almost-sure termination. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 323–337. Springer, Heidelberg (2005)
5. Bradley, A.R., Sipma, H.B., Manna, Z.: Termination of polynomial programs. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 113–129. Springer, Heidelberg (2005)
6. Clarke, E., Donzé, A., Legay, A.: Statistical model checking of mixed-analog circuits with an application to a third order $\Delta - \Sigma$ modulator. In: Chockler, H., Hu, A.J. (eds.) HVC 2008. LNCS, vol. 5394, pp. 149–163. Springer, Heidelberg (2009)
7. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using nonlinear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
8. Colón, M., Sipma, H.: Synthesis of linear ranking functions. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 67–81. Springer, Heidelberg (2001)
9. Cousot, P.: Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 1–24. Springer, Heidelberg (2005)

10. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among the variables of a program. In: POPL 19878, pp. 84–97 (January 1978)
11. Cousot, P., Monerau, M.: Probabilistic abstract interpretation. In: Seidl, H. (ed.) ESOP 2012. LNCS, vol. 7211, pp. 169–193. Springer, Heidelberg (2012)
12. Dubhashi, D., Panconesi, A.: Concentration of Measure for the Analysis of Randomized Algorithms. Cambridge University Press (2009)
13. Esparza, J., Gaiser, A., Kiefer, S.: Proving termination of probabilistic programs using patterns. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 123–138. Springer, Heidelberg (2012)
14. Foster, F.: On the stochastic matrices associated with certain queuing processes. Annals Mathematical Statistics 24(3), 355–360 (1953)
15. Gulwani, S., Srivastava, S., Venkatesan, R.: Program analysis as constraint solving. In: PLDI, pp. 281–292. ACM (2008)
16. Hermanns, H., Wachter, B., Zhang, L.: Probabilistic CEGAR. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 162–175. Springer, Heidelberg (2008)
17. Katoen, J.-P., McIver, A.K., Meinicke, L.A., Morgan, C.C.: Linear-invariant generation for probabilistic programs: In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 390–406. Springer, Heidelberg (2010)
18. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: probabilistic model checking for performance and reliability analysis. SIGMETRICS Perf. Eval. Review 36(4), 40–45 (2009)
19. Lassaigne, R., Peyronnet, S.: Probabilistic verification and approximation. Annals of Pure and Applied Logic 152(1-3), 122–131 (2008)
20. McIver, A., Morgan, C.: Abstraction, Refinement and Proof for Probabilistic Systems. Monographs in Computer Science. Springer (2004)
21. McIver, A.K., Morgan, C.: Developing and reasoning about probabilistic programs in pGCL. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 123–155. Springer, Heidelberg (2006)
22. Miné, A.: A new numerical abstract domain based on difference-bound matrices. In: Danvy, O., Filinski, A. (eds.) PADO 2001. LNCS, vol. 2053, pp. 155–172. Springer, Heidelberg (2001)
23. Monniaux, D.: Abstract interpretation of programs as markov decision processes. Sci. Comput. Program. 58(1-2), 179–205 (2005)
24. Motwani, R., Raghavan, P.: Randomized Algorithms. Cambridge University Press (1995)
25. Müller-Olm, M., Seidl, H.: Precise interprocedural analysis through linear algebra. In: POPL, pp. 330–341. ACM (2004)
26. Platzer, A.: Differential dynamic logic for hybrid systems. J. Autom. Reasoning 41(2), 143–189 (2008)
27. Platzer, A.: Stochastic differential dynamic logic for stochastic hybrid programs. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 446–460. Springer, Heidelberg (2011)
28. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
29. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society (2004)
30. Rodriguez-Carbonell, E., Kapur, D.: Automatic generation of polynomial loop invariants: Algebraic foundations. In: Proc. ISSAC, Spain (2004)
31. Sankaranarayanan, S., Sipma, H., Manna, Z.: Non-linear loop invariant generation using Gröbner bases. In: POPL, pp. 318–330. ACM Press (2004)
32. Williams, D.: Probability with Martingales (Cambridge Mathematical Textbooks). Cambridge University Press (February 1991)
33. Younes, H.L.S., Simmons, R.G.: Statistical probabilitistic model checking with a focus on time-bounded properties. Information & Computation 204(9), 1368–1409 (2006)

Polynomial-Time Verification of PCTL Properties of MDPs with Convex Uncertainties

Alberto Puggelli, Wenchao Li, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia

Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
{puggelli,wenchaol,alberto,sseshia}@eecs.berkeley.edu

Abstract. We address the problem of verifying Probabilistic Computation Tree Logic (PCTL) properties of Markov Decision Processes (MDPs) whose state transition probabilities are only known to lie within uncertainty sets. We first introduce the model of Convex-MDPs (CMDPs), i.e., MDPs with convex uncertainty sets. CMDPs generalize Interval-MDPs (IMDPs) by allowing also more expressive (convex) descriptions of uncertainty. Using results on strong duality for convex programs, we then present a PCTL verification algorithm for CMDPs, and prove that it runs in time polynomial in the size of a CMDP for a rich subclass of convex uncertainty models. This result allows us to lower the previously known algorithmic complexity upper bound for IMDPs from co-NP to PTIME. We demonstrate the practical effectiveness of the proposed approach by verifying a consensus protocol and a dynamic configuration protocol for IPv4 addresses.

1 Introduction

Stochastic models such as Discrete-Time Markov Chains (DTMCs) [1] and Markov Decision Processes (MDPs) [2] are used to formally represent systems that exhibit probabilistic behaviors. These systems need *quantitative* analysis [3] to answer questions such as "what is the probability that a request will be eventually served?". Properties of these systems can be expressed and analyzed using logics such as Probabilistic Computation Tree Logic (PCTL) [4] — a probabilistic logic derived from CTL — as well as techniques for probabilistic model checking [5]. These methods often rely on deriving a probabilistic model of the underlying process, hence the formal guarantees they provide are only as good as the estimated model. In a real setting, these estimations are affected by uncertainties due, for example, to measurement errors or approximation of the real system by mathematical models.

Interval-valued Discrete-Time Markov Chains (IDTMCs) have been introduced to capture modeling uncertainties [6]. IDTMCs are DTMC models where each transition probability lies within a closed interval. Two semantic interpretations have been proposed for IDTMCs [7]: Uncertain Markov Chains (UMCs) and Interval Markov Decision Processes (IMDPs). An UMC is interpreted as a family of DTMCs, where each member is a DTMC whose transition probabilities lie within the interval range given in the UMC. In IMDPs, the uncertainty is resolved through non-determinism. Each time a state is visited, a transition distribution within the interval is adversarially picked,

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 527–542, 2013.

and a probabilistic step is taken accordingly. Thus, IMDPs model a non-deterministic choice made from a set of (possibly uncountably many) choices. In this paper we do not consider UMCs and focus on IMDPs.

An upper-bound on the complexity of model checking PCTL properties on IMDPs was previously shown to be co-NP [8]. This result relies on the construction of an equivalent MDP that encodes all behaviors of the IMDP. For each state in the new MDP, the set of transition probabilities is equal to the Basic Feasible Solutions (BFS) of the set of inequalities specifying the transition probabilities of the IMDP. Since the number of BFS is exponential in the number of states in the IMDP, the equivalent MDP can have size exponential in the size of the IMDP. In this paper, we describe a *polynomial-time algorithm* (in both size of the model and size of the formula) based on Convex Programming (CP) for the same fragment of PCTL considered in [7, 8] (the *Bounded Until* operator is disallowed). This shows that the problem is in the complexity class PTIME. With *Bounded Until*, the time complexity of our algorithm only increases to pseudo-polynomial in the maximum integer time bound.

An interval model of uncertainty may appear to be the most intuitive. However, there are significant advantages in accommodating also more expressive (and less pessimistic) uncertainty models. In [9], a financial portfolio optimization case-study is analyzed in which uncertainty arises from estimating the asset return rates. The authors claim that the interval model is too conservative in this scenario, because it would suggest to invest the whole capital into the asset with the smallest worst-case return. The ellipsoidal model proposed in that paper returns instead the more profitable strategy of spreading the capital across multiple assets. Further, depending on the field, researchers use different models to represent uncertainty. Maximum likelihood models are often used, for example, to estimate chemical reaction parameters [10]. To increase modeling expressiveness, we introduce the model of *Convex-MDP (CMDP)*, i.e., an MDP whose state transition probabilities are only known to lie within convex uncertainty sets. The proposed algorithms can be extended to verify CMDPs for all the models of uncertainty that satisfy a technical condition introduced later in the paper, while maintaining the same complexity results proven for IMDPs. This condition is not a limitation in practical scenarios, and we show that all the models in the wide and relevant class of convex uncertainty sets introduced in [11] (e.g. interval, ellipsoidal and likelihood models) satisfy it. Heterogeneous models of uncertainty can then be used within the same CMDP to represent different sources of uncertainty. We also note that the complexity results presented in [7] and [8] cannot be trivially extended to verifying CMDPs. This is because BFS are not defined for generic convex inequalities, so the construction of an equivalent MDP would not be possible. The complexity results are compared in Table 1.

To summarize, the contributions of this paper are as follows.

1. We give a polynomial-time algorithm for model checking PCTL properties (without *Bounded Until*) on IMDPs. This improves the co-NP result in [8] to PTIME.
2. We extend the algorithm to full PCTL and show that its time complexity becomes pseudo-polynomial in the maximum integer bound in *Bounded Until*.
3. We show that our complexity results extend to Convex-MDPs (CMDPs) for a wide and expressive subclass of the convex models of uncertainty.

Table 1. Known Upper-Bound on the Complexity of PCTL Model Checking

Model	DTMC [4]	IMDP [8]	IMDP/CMDP [ours]
Complexity	PTIME	co-NP	PTIME

4. We demonstrate the relevance of our approach with case studies, where a small uncertainty in the probability transitions indeed yields a significant change in the verification results.

An extended version of the paper with details of all verification algorithms and proofs of correctness is available [12].

The paper is organized as follows. Section 2 gives background on MDPs, PCTL, and the analyzed uncertainty models. Section 3 presents related work. Section 4 gives an overview of the proposed approach. In Section 5, we describe the proposed algorithm in detail and prove the PTIME complexity result. Section 6 describes two case studies, and we conclude and discuss future directions in Section 7.

2 Preliminaries

Definition 2.1. *A Probability Distribution (PD) over a finite set Z of cardinality n is a vector $\mu \in \mathbb{R}^n$ satisfying $\mathbf{0} \leq \mu \leq \mathbf{1}$ and $\mathbf{1}^T\mu = 1$. The element $\mu[i]$ represents the probability of realization of event z_i. We call $Dist(Z)$ the set of distributions over Z.*

2.1 Convex Markov Decision Process (CMDP)

Definition 2.2. *A CMDP is a tuple $\mathcal{M}_\mathcal{C} = (S, S_0, A, \Omega, \mathcal{F}, \mathcal{A}, \mathcal{X}, L)$, where S is a finite set of states of cardinality $N = |S|$, S_0 is the set of initial states, A is a finite set of actions ($M = |A|$), Ω is a finite set of atomic propositions, $\mathcal{F}$ is a finite set of convex sets of transition PDs, $\mathcal{A} : S \to 2^A$ is a function that maps each state to the set of actions available at that state, $\mathcal{X} = S \times A \to \mathcal{F}$ is a function that associates to state s and action a the corresponding convex set $\mathcal{F}_s^a \in \mathcal{F}$ of transition PDs, and $L : S \to 2^\Omega$ is a labeling function.*

The set $\mathcal{F}_s^a = Dist_s^a(S)$ represents the uncertainty in defining a transition distribution for $\mathcal{M}_\mathcal{C}$ given state s and action a. We call $\mathbf{f}_s^a \in \mathcal{F}_s^a$ an observation of this uncertainty. Also, $\mathbf{f}_s^a \in \mathbb{R}^N$ and we can collect the vectors $\mathbf{f}_s^a, \forall s \in S$ into an observed transition matrix $F^a \in \mathbb{R}^{N \times N}$. Abusing terminology, we call $\mathcal{F}^a$ the uncertainty set of the transition matrices, and $F^a \in \mathcal{F}^a$. $\mathcal{F}_s^a$ is interpreted as the row of $\mathcal{F}^a$ corresponding to state s. Finally, $f_{s_i s_j}^a = \mathbf{f}_{s_i}^a[j]$ is the observed probability of transitioning from s_i to s_j when action a is selected.

A transition between state s to state s' in a CMDP occurs in three steps. First, an action $a \in \mathcal{A}(s)$ is chosen. The selection of a is nondeterministic. Secondly, an observed PD $\mathbf{f}_s^a \in \mathcal{F}_s^a$ is chosen. The selection of $\mathbf{f}_s^a$ models uncertainty in the transition. Lastly, a successor state s' is chosen randomly, according to the transition PD $\mathbf{f}_s^a$.

A path π in $\mathcal{M}_\mathcal{C}$ is a finite or infinite sequence of the form $s_0 \xrightarrow{f_{s_0 s_1}^{a_0}} s_1 \xrightarrow{f_{s_1 s_2}^{a_1}}, \cdots$, where $s_i \in S$, $a_i \in \mathcal{A}(s_i)$ and $f_{s_i, s_{i+1}}^{a_i} > 0 \; \forall i \geq 0$. We indicate with Π_{fin} (Π_{inf}) the

set of all finite (infinite) paths of $\mathcal{M}_\mathcal{C}$. $\pi[i]$ is the i^{th} state along the path and, for finite paths, $last(\pi)$ is the last state visited in $\pi \in \Pi_{fin}$. $\Pi_s = \{\pi \mid \pi[0] = s\}$ is the set of paths starting in state s.

To model uncertainty in state transitions, we make the following assumptions:

Assumption 2.1. $\mathcal{F}^a$ can be factored as the Cartesian product of its rows, i.e., its rows are uncorrelated. Formally, for every $a \in A$, $\mathcal{F}^a = \mathcal{F}^a_{s_0} \times \cdots \times \mathcal{F}^a_{s_{N-1}}$. In [11] this assumption is referred to as *rectangular uncertainty*.

Assumption 2.2. If the probability of a transition is zero (non-zero) for at least one PD in the uncertainty set, then it is zero (non-zero) for all PDs.
Formally, $\exists \mathbf{f}^a_s \in \mathcal{F}^a_s : f^a_{ss'} = (\neq)0 \implies \forall \mathbf{f}^a_s \in \mathcal{F}^a_s : f^a_{ss'} = (\neq)0$.
The assumption guarantees the correctness of the preprocessing verification routines used later in the paper, which rely on state reachability of the MDP underlying graph.

We determine the size $\mathcal{R}$ of the CMDP $\mathcal{M}_\mathcal{C}$ as follows. $\mathcal{M}_\mathcal{C}$ has N states, $O(M)$ actions per state and $O(N^2)$ transitions for each action. Let D^a_s denote the number of constraints required to express the rectangular uncertainty set $\mathcal{F}^a_s$ (e.g. $D^a_s = O(2N)$ for the interval model, to express the upper and lower bounds of the transition probabilities from state s to all states $s' \in S$), and $D = \max_{s \in S, a \in A} D^a_s$. The overall size of $\mathcal{M}_\mathcal{C}$ is thus $\mathcal{R} = O(N^2M + NMD)$.

In order to analyze *quantitative* properties of CMDPs, we need a probability space over infinite paths [13]. However, a probability space can only be constructed once nondeterminism and uncertainty have been resolved. We call each possible resolution of nondeterminism an *adversary*, which chooses an action in each state of $\mathcal{M}_\mathcal{C}$.

Definition 2.3. Adversary. *A randomized adversary for $\mathcal{M}_\mathcal{C}$ is a function $\alpha = \Pi_{fin} \times A \to [0,1]$, with $\sum_{\mathcal{A}(last(\pi))} \alpha(\pi, a) = 1$, and $a \in \mathcal{A}(last(\pi))$ if $\alpha(\pi, a) > 0$. We call Adv the set of all adversaries α of $\mathcal{M}_\mathcal{C}$.*

Conversely, we call a *nature* each possible resolution of uncertainty, i.e., a nature chooses a transition PD for each state and action of $\mathcal{M}_\mathcal{C}$.

Definition 2.4. Nature. *Given action $a \in A$, a randomized nature is the function $\eta^a : \Pi_{fin} \times Dist(S) \to [0,1]$ with $\int_{\mathcal{F}^a_{last(\pi)}} \eta^a(\pi, \mathbf{f}^a_s) = 1$, and $\mathbf{f}^a_s \in \mathcal{F}^a_{last(\pi)}$ if $\eta^a(\pi, \mathbf{f}^a_s) > 0$. We call Nat the set of all natures η^a of $\mathcal{M}_\mathcal{C}$.*

An adversary α (nature η^a) is memoryless if it depends only on $last(\pi)$. Also, α (η^a) is deterministic if $\alpha(\pi, a) = 1$ for some $a \in \mathcal{A}(last(\pi))$ ($\eta^a(\pi, f^a_s) = 1$ for some $\mathbf{f}^a_s \in \mathcal{F}^a_{last(\pi)}$).

2.2 Models of Uncertainty

We only consider CMDPs whose transition PDs lie in uncertainty sets that satisfy Assumption 5.1 (introduced later for ease of presentation). This assumption holds for all the uncertainty models analyzed in [11]. We report results for the interval, likelihood and ellipsoidal models. Results for the entropy model are available in [12].

Interval Model. Intervals commonly describe uncertainty in transition matrices:

$$\mathcal{F}_s^a = \{\mathbf{f}_s^a \in \mathbb{R}^N \mid \mathbf{0} \leq \underline{\mathbf{f}}_s^a \leq \mathbf{f}_s^a \leq \overline{\mathbf{f}}_s^a \leq \mathbf{1}, \mathbf{1}^T\mathbf{f}_s^a = 1\} \tag{1}$$

where $\underline{\mathbf{f}}_s^a, \overline{\mathbf{f}}_s^a \in \mathbb{R}^N$ are the element-wise lower and upper bounds of $\mathbf{f}$. This model is suitable when the matrix components are individually estimated by statistical data. An IMDP is a CMDP in which all uncertainties are described using the interval model and with the number of available actions $M = 1$.

Likelihood Model. This model is appropriate when the transition probabilities are determined experimentally. The transition frequencies associated to action $a \in A$ are collected in matrix H^a. Uncertainty in each row of H^a can be described by the likelihood region [14]:

$$\mathcal{F}_s^a = \{\mathbf{f}_s^a \in \mathbb{R}^N \mid \mathbf{f}_s^a \geq \mathbf{0}, \mathbf{1}^T\mathbf{f}_s^a = 1, \textstyle\sum_{s'} h_{ss'}^a \log(f_{ss'}^a) \geq \beta_s^a\} \tag{2}$$

where $\beta_s^a < \beta_{s,max}^a = \sum_{s'} h_{ss'}^a \log(h_{ss'}^a)$ represents the uncertainty level. Likelihood regions are less conservative uncertainty representations than intervals, which arise from projections of the uncertainty region onto each row component.

Ellipsoidal Model. Ellipsoidal models can be seen as a second-order approximation of the likelihood model [11]. Formally:

$$\mathcal{F}_s^a = \{\mathbf{f}_s^a \in \mathbb{R}^N \mid \mathbf{f}_s^a \geq \mathbf{0}, \mathbf{1}^T\mathbf{f}_s^a = 1, \|R_s^a\,(\mathbf{f}_s^a - \mathbf{h}_s^a)\,\|_2 \leq 1, R_s^a \succ 0\} \tag{3}$$

where matrix R_s^a represents an ellipsoidal approximation of the likelihood Region (2).

Remark 2.1. Each set $\mathcal{F}_s^a$ within the same CMDP can be expressed with a different uncertainty model to represent different sources of uncertainty.

2.3 Probabilistic Computation Tree Logic (PCTL)

We use PCTL, a probabilistic logic derived from CTL which includes a probabilistic operator P [4], to express properties of CMDPs. The syntax of this logic is:

$$\phi ::= True \mid \omega \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid P_{\bowtie p}\,[\psi] \qquad \text{state formulas}$$
$$\psi ::= \mathcal{X}\phi \mid \phi_1\,\mathcal{U}^{\leq k}\phi_2 \mid \phi_1\,\mathcal{U}\phi_2 \qquad \text{path formulas}$$

where $\omega \in \Omega$ is an atomic proposition, $\bowtie \in \{\leq, <, \geq, >\}$, $p \in [0,1]$ and $k \in \mathbb{N}$.

Table 2. PCTL semantics for CMDP

$s \models$ True		$s \models \omega$	iff $\omega \in L(s)$
$s \models \neg\phi$	iff $s \not\models \phi$	$s \models \phi_1 \wedge \phi_2$	iff $s \models \phi_1 \wedge s \models \phi_2$
$s \models P_{\bowtie p}\,[\psi]$	iff $Prob\,(\{\pi \in \Pi_s(\alpha, \eta^a) \mid \pi \models \psi\}) \bowtie p$	$\forall \alpha \in Adv$ and $\eta^a \in Nat$	
$\pi \models \mathcal{X}\phi$	iff $\pi[1] \models \phi$		
$\pi \models \phi_1\,\mathcal{U}^{\leq k}\phi_2$	iff $\exists i \leq k \mid \pi[i] \models \phi_2 \wedge \forall j < i\ \pi[j] \models \phi_1$		
$\pi \models \phi_1\,\mathcal{U}\phi_2$	iff $\exists k \geq 0 \mid \pi \models \phi_1\,\mathcal{U}^{\leq k}\phi_2$		

Path formulas ψ use the *Next* $(\mathcal{X})$, *Bounded Until* $\left(\mathcal{U}^{\leq k}\right)$ and *Unbounded Until* $(\mathcal{U})$ operators. These formulas are evaluated over paths and only allowed as parameters to the $P_{\bowtie p}[\psi]$ operator. The size $\mathcal{Q}$ of a PCTL formula is defined as the number of Boolean connectives plus the number of temporal operators in the formula. For the *Bounded Until* operator, we denote separately the maximum time bound that appears in the formula as k_{max}. Probabilistic statements about MDPs typically involve universal quantification over adversaries $\alpha \in Adv$. With uncertainties, for each action a selected by adversary α, we will further quantify across nature $\eta^a \in Nat$ to compute the worst case condition within the action range of η^a, i.e., the uncertainty set $\mathcal{F}_s^a$. We define $P_s(\alpha, \eta^a)[\psi] \triangleq Prob\left(\{\pi \in \Pi_s(\alpha, \eta^a) \mid \pi \models \psi\}\right)$ the probability of taking a path $\pi \in \Pi_s$ that satisfies ψ under adversary α and nature η^a. If α and η^a are Markov deterministic in state s, we write $P_s(a, \mathbf{f}_s^a)$, where a and $\mathbf{f}_s^a$ are the action and resolution of uncertainty that are deterministically chosen at each execution step by α and η^a. $P_s^{max}[\psi]$ ($P_s^{min}[\psi]$) denote the maximum (minimum) probability $P_s(\alpha, \eta^a)[\psi]$ across all adversaries $\alpha \in Adv$ and natures $\eta^a \in Nat$, and the vectors $\mathbf{P}^{max}[\psi], \mathbf{P}^{min}[\psi] \in \mathbb{R}^N$ collect these probabilities $\forall s \in S$. The semantics of the logic is reported in Table 2, where we write $\models$ instead of $\models_{Adv,Nat}$ for simplicity.

For ease of computation, we would like to consider only memoryless and deterministic adversaries and natures to compute *quantitative* probabilities, i.e., solve:

$$P_s^{max}[\psi] = \max_{a \in \mathcal{A}(s)} \max_{\mathbf{f}_s^a \in \mathcal{F}_s^a} P_s(a, \mathbf{f}_s^a)[\psi] \qquad P_s^{min}[\psi] = \min_{a \in \mathcal{A}(s)} \min_{\mathbf{f}_s^a \in \mathcal{F}_s^a} P_s(a, \mathbf{f}_s^a)[\psi] \quad (4)$$

We extend a result from [15] to prove that this is possible (see [12] for the proof).

Proposition 2.1. *Given a CMDP $\mathcal{M}_\mathcal{C}$ and a target state $s_t \in S$, there always exist deterministic and memoryless adversaries and natures for $\mathcal{M}_\mathcal{C}$ that achieve the maximum (minimum) probabilities of reaching s_t, if A is finite and the inner optimization in Problem (4) always attains its optimum $\sigma_s^*(a)$ over the sets $\mathcal{F}_s^a, \forall s \in S, \forall a \in \mathcal{A}(s)$, i.e., there exists a finite feasible $\mathbf{f}_s^a \in \mathcal{F}_s^a$ such that $P_s(a, \mathbf{f}_s^a)[\psi] = \sigma_s^*(a)$.*

The verification algorithm V determines whether a state $s \in S_0$ is (is not) contained in the set $Sat(\phi) = \{s \in S \mid s \models \phi\}$. We define the following properties for V:

Definition 2.5. Soundness *(*Completeness*). Algorithm V is sound (complete) if:*

$$s \in Sat_V(\phi) \Rightarrow s \in Sat(\phi) \qquad (s \notin Sat_V(\phi) \Rightarrow s \notin Sat(\phi))$$

where $Sat_V(\phi)$ ($Sat(\phi)$) is the computed (actual) satisfaction set.

Algorithms to verify non-probabilistic formulas are sound and complete, because they are based on reachability analysis over the finite number of states of $\mathcal{M}_\mathcal{C}$ [16]. Conversely, we will show in Section 5 that algorithms to verify probabilistic formulas $\phi = P_{\bowtie p}[\psi]$ in the presence of uncertainties require solving convex optimization problems over the set $\mathbb{R}$ of the real numbers. Optima of these problems can be arbitrary real numbers, so, in general, they can be computed only to within a desired accuracy ϵ_d. We consider an algorithm to be sound and complete if the error in determining the satisfaction probabilities of ϕ is bounded by such a parameter ϵ_d, since the returned result will still be accurate enough in most settings.

3 Related Work

Probabilistic model checking tools such as PRISM [5] have been used to analyze a multitude of applications, from communication protocols and biological pathways to security problems. In this paper, we further consider *uncertainties* in the probabilistic transitions of the MDP for model checking PCTL specifications. Prior work [6–8, 17] in similar verification problems also dealt with uncertainties in the probabilistic transitions. However, they considered only interval models of uncertainty, while we incorporate more expressive models such as ellipsoidal and likelihood. Further, we consider nature as adversarial and study how it affects the MDP execution in the worst case. The developers of PARAM [18] consider instead uncertainties as possible values that parameters in the model can take, and synthesize the optimal parameter values to maximize the satisfaction probability of a given PCTL specification.

We improve the previously best-known complexity result of co-NP in [8] to PTIME, for the fragment of PCTL without $\mathcal{U}^{\leq k}$. For the full PCTL syntax, our algorithm runs in $O(poly(\mathcal{R}) \times \mathcal{Q} \times k_{max})$ time, where k_{max} is the maximum bound in $\mathcal{U}^{\leq k}$. This result is pseudo-polynomial in k_{max}, i.e., polynomial (exponential) if k_{max} is counted in its unary (binary) representation. Conversely, classical PCTL model checking for DTMCs [4] runs in time polynomial in k_{max} counted in its binary representation. The difference stems from the computation of the set $Sat\left(P_{\bowtie p}\left[\phi_1\ \mathcal{U}^{\leq k}\phi_2\right]\right)$. For (certain) MDPs, this computation involves raising the transition matrices $F^a, \forall a \in A$ to the k^{th} power, to model the evolution of the system in k steps. With uncertainties, we cannot do matrix exponentiation, because $F^a \in \mathcal{F}^a$ might change at each step. However, both $\mathcal{Q}$ and k_{max} are typically small in practical applications [19], so the dominant factor for runtime is the size of the model $\mathcal{R}$. We note that the complexity results of [7] and [8] can be extended to the PCTL with $\mathcal{U}^{\leq k}$.

The convex uncertainty models [11] analyzed in this paper have been considered recently in the robust control literature. In [20], an algorithm is given to synthesize a robust optimal controller for an MDP to satisfy a Linear Temporal Logic (LTL) specification where only one probabilistic operator is allowed. Their technique first converts the LTL specification to a Rabin automaton (which is worst-case doubly exponential in the size of the LTL formula), and composes it with the MDP. Robust dynamic programming is then used to solve for the optimal control policy. We consider PCTL, which allows nested probability operators, and propose an algorithm which is polynomial both in the size of the model and of the formula.

The robustness of PCTL model checking has been analyzed [21] based on the notion of an Approximate Probabilistic Bisimulation (APB) tailored to the finite-precision approximation of a numerical model. We instead verify MDPs whose transition probabilities are affected by uncertainties due to estimation errors or imperfect information about the environment.

4 Probabilistic Model Checking with Uncertainties

We define the problem under analysis, and give an overview the proposed approach to solve it.

PCTL model checking with uncertainties. **Given** a Markov Decision Process model with convex uncertainties $\mathcal{M}_\mathcal{C}$ of size $\mathcal{R}$ and a PCTL formula ϕ of size $\mathcal{Q}$ over a set of atomic propositions Ω, **verify** ϕ over the uncertainty sets $\mathcal{F}_s^a \in \mathcal{F}$ of $\mathcal{M}_\mathcal{C}$.

As in verification of CTL [22], the algorithm traverses bottom-up the parse tree for ϕ, recursively computing the set $Sat(\phi')$ of states satisfying each sub-formula ϕ'. At the end of the traversal, the algorithm computes the set of states satisfying ϕ and it determines if $s \models \phi$ by checking if $s \in Sat\,(\phi)$. For the non-probabilistic PCTL operators, the satisfying states are computed as: $Sat\,(True) = S$, $Sat(\omega) = \{s \in S \mid \omega \in L(s)\}$, $Sat(\neg\phi) = S \setminus Sat(\phi)$ and $Sat(\phi_1 \wedge \phi_2) = Sat(\phi_1) \cap Sat(\phi_2)$. For the probabilistic operator $P \bowtie [\psi]$, we compute:

$$Sat\,(P_{\triangleleft p}\,[\psi])=\{s \in S \mid P_s^{max}(\psi) \triangleleft p\}\,,\quad Sat\,(P_{\triangleright p}\,[\psi])=\left\{s \in S \mid P_s^{min}(\psi) \triangleright p\right\} \quad (5)$$

We propose polynomial-time routines to compute Sets 5 for MDPs whose transition matrices F^a are only known to lie within convex uncertainty sets $\mathcal{F}^a$, $\forall a \in A$.

Using Proposition 2.1, the proposed routines encode the transitions of $\mathcal{M}_\mathcal{C}$ under the sets of deterministic and memoryless adversaries and natures into convex programs and solve them. From the returned solution, it is then possible to determine the *quantitative* satisfaction probabilities $P_s^{max}[\psi]$ (or $P_s^{min}[\psi]$) $\forall s \in S$, which get compared in linear time to the threshold p to compute the set $Sat\,(P_{\bowtie p}\,[\psi])$. To prove the polynomial-time complexity of the model-checking algorithm, we use the following key result from convex theory [23].

Proposition 4.1. *Given the convex program:*

$$\begin{aligned} &\underset{\mathbf{x}}{min}\ f_0(\mathbf{x}) \\ &s.t.\ f_i(\mathbf{x}) \leq 0 \qquad\qquad i = 1, \cdots, m \end{aligned}$$

with $\mathbf{x} \in \mathbb{R}^n$ *and* $f_i, i = 0, \cdots, m$ *convex functions, the optimum* σ^* *can be found to within* $\pm\epsilon_d$ *in time complexity polynomial in the problem size* (n, m) *and* $\log(1/\epsilon_d)$.

We are now ready to state the main contribution of this paper:

Theorem 4.1. Complexity of PCTL Model-Checking for CMDPs.

1. *The problem of verifying if a CMDP* $\mathcal{M}_\mathcal{C}$ *of size* $\mathcal{R}$ *satisfies a PCTL formula* ϕ *without* $\mathcal{U}^{\leq k}$ *is in PTIME.*
2. *A formula* ϕ' *with* $\mathcal{U}^{\leq k}$ *can be verified with time complexity* $O\,(poly(\mathcal{R}) \times \mathcal{Q}' \times k_{max})$*, i.e., pseudo-polynomial in the maximum time bound* k_{max} *of* $\mathcal{U}^{\leq k}$.

Sketch of proof. The proof is constructive. Our verification algorithm parses ϕ in time linear in the size $\mathcal{Q}$ of ϕ [22], computing the satisfiability set of each operator in ϕ. For the non-probabilistic operators, satisfiability sets can be computed in time polynomial in $\mathcal{R}$ using set operations, i.e., set inclusion, complementation and intersection. For the probabilistic operator, we leverage Proposition 4.1 and prove that the proposed verification routines: 1) solve a number of convex problems polynomial in $\mathcal{R}$; 2) generate these convex programs in time polynomial in $\mathcal{R}$. It thus follows that the overall algorithm runs in time polynomial in $\mathcal{R}$ and in the size of ϕ. The correctness and time-complexity for formulas involving the *Unbounded Until* operator are formalized in Lemma 5.1. Results regarding the *Next* and *Bounded Until* operator can be found in [12]. □

5 Verification Routines

We detail the routine to verify the *Unbounded Until* operator. Routines to verify the *Next* and *Bounded Until* operators can be found in the extended version [12].

5.1 *Unbounded Until* Operator

We verify $\phi = P_{\lhd p}[\phi_1 \mathcal{U} \phi_2]$ on a CMDP of size $\mathcal{R}$. First, the sets $S^{yes} \triangleq Sat\left(P_{\geq 1}[\phi_1 \mathcal{U} \phi_2]\right)$, $S^{no} \triangleq Sat\left(P_{\leq 0}[\phi_1 \mathcal{U} \phi_2]\right)$ and $S^{?} = S \setminus (S^{no} \cup S^{yes})$ are precomputed in time polynomial in $\mathcal{R}$ using reachability routines over the CMDP underlying graph [16]. Second, Equation (4) is evaluated for all $s \in S$ using the Convex Programming procedure described next. Finally, the computed probabilities are compared to p.

Convex Programming Procedure (CP). We start from the classical LP formulation to solve the problem without the presence of uncertainty [16]:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{x^T 1} \\ &\text{s.t. } x_s = 0;\ x_s = 1; && \forall s \in S^{no};\ s \in S^{yes}; \\ &\quad x_s \geq \mathbf{x^T f_s^a} && \forall s \in S^{?}, \forall a \in \mathcal{A}(s) \end{aligned} \tag{6}$$

where $\mathbf{P}^{max}[\phi_1 \mathcal{U} \phi_2] = \mathbf{x}^*$ is computed solving only one LP. Problem (6) has N unknowns and $N - Q + MQ$ constraints, where $Q = |S^{?}| = O(N)$, so its size is polynomial in $\mathcal{R}$.

Proposition 2.1 allows us to rewrite Problem (6) in the uncertain scenario as:

$$\begin{aligned} &\min_{\mathbf{x}} \mathbf{x^T 1} \\ &\text{s.t. } x_s = 0;\ x_s = 1; && \forall s \in S^{no};\ \forall s \in S^{yes}; \\ &\quad x_s \geq \max_{\mathbf{f_s^a} \in \mathcal{F}_s^a} \left(\mathbf{x^T f_s^a}\right) && \forall s \in S^{?}, \forall a \in \mathcal{A}(s) \end{aligned} \tag{7}$$

i.e., we maximize the lower bound on x_s across the nature action range. The decision variable of the inner problem is $\mathbf{f}_s^a$ and its optimal value $\sigma^*(\mathbf{x})$ is parameterized in the outer problem decision variable $\mathbf{x}$. Problem (7) can be written in convex form for an arbitrary uncertainty model by replacing the last constraint with one constraint for each point in $\mathcal{F}_s^a$. However, this approach results in infinite constraints if the set $\mathcal{F}_s^a$ contains infinitely many points, as in the cases considered in the paper. We solve this difficulty using duality, which allows us to rewrite Problem (7) with a number of constraints polynomial in $\mathcal{R}$. We start by replacing the primal inner problem in the outer Problem (7) with its dual $\forall s \in S^{?}$ and $\forall a \in \mathcal{A}(s)$:

$$\sigma_s^a(\mathbf{x}) = \max_{\mathbf{f}_s^a \in \mathcal{F}_s^a} \mathbf{x}^T \mathbf{f}_s^a \qquad \Rightarrow \qquad d_s^a(\mathbf{x}) = \min_{\lambda_{\mathbf{s}}^{\mathbf{a}} \in \mathcal{D}_s^a} g(\lambda_{\mathbf{s}}^{\mathbf{a}}, \mathbf{x}) \tag{8}$$

where $\lambda_{\mathbf{s}}^{\mathbf{a}}$ is the (vector) Lagrange multiplier and $\mathcal{D}_s^a$ is the feasibility set of the dual. In the dual, the decision variable is $\lambda_{\mathbf{s}}^{\mathbf{a}}$ and its optimal value $d_s^a(\mathbf{x})$ is parameterized in $\mathbf{x}$. The dual function $g(\lambda_s^a, \mathbf{x})$ and the set $\mathcal{D}_s^a$ are convex by construction in λ_s^a for arbitrary uncertainty models, so the dual is convex. Further, since also the primal is convex, strong duality holds, i.e., $\sigma_s^a = d_s^a$, $\forall \mathbf{x} \in \mathbb{R}^N$, because the primal satisfies Slater's condition [24] for any non-trivial uncertainty set $\mathcal{F}_s^a$. Any dual solution overestimates

the primal solution. When substituting the primals with the duals in Problem (7), we drop the inner optimization operators because the outer optimization operator will find the least overestimates, i.e., the dual solutions $d_s^a, \forall s \in S, a \in \mathcal{A}(s)$, to minimize its cost function. We get the CP formulation:

$$\begin{aligned}
&\min_{\mathbf{x}} \mathbf{x^T 1} && \min_{\mathbf{x},\lambda} \mathbf{x^T 1} && \\
&\text{s.t. } x_s = 0;\ x_s = 1; && \text{s.t. } x_s = 0;\ x_s = 1; && \forall s \in S^{no}; \forall s \in S^{yes}; && \text{(9a)}\\
&\quad x_s \geq \min_{\lambda_s^a \in \mathcal{D}_s^a} g\left(\lambda_s^a, \mathbf{x}\right) && \Rightarrow \quad x_s \geq g\left(\lambda_s^a, \mathbf{x}\right); && \forall s \in S^{?}, \forall a \in \mathcal{A}(s); && \text{(9b)}\\
& && \quad \lambda_s^a \in \mathcal{D}_s^a && \forall s \in S^{?}, \forall a \in \mathcal{A}(s) && \text{(9c)}
\end{aligned}$$

The decision variables of Problem (9) are both $\mathbf{x}$ and λ_s^a, so the CP formulation is convex only if the dual function $g(\lambda_s^a, \mathbf{x})$ is jointly convex in λ_s^a and $\mathbf{x}$. While this condition cannot be guaranteed for arbitrary uncertainty models, we prove constructively that it holds for the ones considered in the paper. For example, for the interval model, Problem (9) reads:

$$\begin{aligned}
&\min_{\mathbf{x},\lambda_\mathbf{s}^\mathbf{a}} \mathbf{x^T 1} && \\
&\text{s.t. } x_s = 0;\ x_s = 1; && \forall s \in S^{no}; \forall s \in S^{yes}; && \\
&\quad x_s \geq \lambda_{1,s}^a - (\underline{\mathbf{f}}_a^s)^T \lambda_{\mathbf{2},\mathbf{s}}^\mathbf{a} + (\overline{\mathbf{f}}_a^s)^T \lambda_{\mathbf{3},\mathbf{s}}^\mathbf{a}; && \forall s \in S^{?}, \forall a \in \mathcal{A}(s); && \text{(10a)}\\
&\quad \mathbf{x} + \lambda_{\mathbf{2},\mathbf{s}}^\mathbf{a} - \lambda_{\mathbf{3},\mathbf{s}}^\mathbf{a} - \lambda_{1,s}^a \mathbf{1} = \mathbf{0}; && \forall s \in S^{?}, \forall a \in \mathcal{A}(s); && \text{(10b)}\\
&\quad \lambda_{\mathbf{2},\mathbf{s}}^\mathbf{a} \geq \mathbf{0},\ \lambda_{\mathbf{3},\mathbf{s}}^\mathbf{a} \geq \mathbf{0} && \forall s \in S^{?}, \forall a \in \mathcal{A}(s) && \text{(10c)}
\end{aligned}$$

which is an LP, so trivially jointly convex in $\mathbf{x}$ and λ_s^a. Analogously, Problem (9) for the ellipsoidal model is a Second-Order Cone Program (SOCP), so again jointly convex in $\mathbf{x}$ and λ_s^a [12]. For the likelihood model, Constraints (10a-10c) become:

$$\begin{aligned}
&x_s \geq \lambda_{1,s}^a - (1+\beta_s^a)\lambda_{2,s}^a + \lambda_{2,s}^a \textstyle\sum_{s'} h_{ss'}^a \log\left(\frac{\lambda_{2,s}^a h_{ss'}^a}{\lambda_{1,s}^a - x_{s'}}\right); && \forall s \in S^{?}, \forall a \in \mathcal{A}(s); && \text{(11a)}\\
&\lambda_{1,s}^a \geq \max_{s' \in S} x_{s'};\ \lambda_{2,s}^a \geq 0 && \forall s \in S^{?}, \forall a \in \mathcal{A}(s) && \text{(11b)}
\end{aligned}$$

We prove its joint convexity in $\mathbf{x}$ and λ_s^a as follows. Constraint (11a) is generated by a primal-dual transformation, so, according to convex theory, it is convex in the dual variables λ_s^a by construction. Convex theory also guarantees that the affine subtraction of $\mathbf{x}$ from $\lambda_{1,s}^a$ preserves convexity, given $\lambda_{1,s}^a \geq \max\ x_{s'}, \forall s \in S$ in Constraint (11b), so we conclude that Problem (11) is convex.

For general CMDPs, we will assume:

Assumption 5.1. Given a CMDP $\mathcal{M}_\mathcal{C}$, for all convex uncertainty sets $\mathcal{F}_s^a \in \mathcal{F}$, the dual function $g(\lambda_s^a, \mathbf{x})$ in Problem (8) is jointly convex in both $\lambda_\mathbf{s}^\mathbf{a}$ and $\mathbf{x}$.

According to Proposition 4.1, Problem (9) can thus be solved in polynomial time. Also, $\mathbf{P}^{max}[\phi_1 \mathcal{U} \phi_2] = \mathbf{x}^*$, so all the satisfaction probabilities can be computed by solving only one convex problem. Finally, we can combine models of uncertainty different from one another within a single CP formulation, since each dual problem is independent from the others according to Assumption 2.1. As an example, if both the interval and ellipsoidal models are used, the overall CP formulation is an SOCP.

Lemma 5.1. *The routine to verify the* Unbounded Until *operator is sound, complete and guaranteed to terminate with algorithmic complexity polynomial in the size $\mathcal{R}$ of $\mathcal{M}_\mathcal{C}$, if $\mathcal{M}_\mathcal{C}$ satisfies Assumption 5.1.*

Proof. The routine solves only one convex program, generated in time polynomial in $\mathcal{R}$ as follows. We formulate Constraints (9b) and (9c) $\forall s \in S^?$ and $a \in \mathcal{A}(s)$, i.e., $O(MQ)$ constraints, where $Q = |S^?| = O(N)$. They are derived from MQ primal-dual transformations as in Equation (8). Each primal problem has N unknowns, $N+1$ constraints to represent the probability simplex and D_s^a constraints to represent the uncertainty set $\mathcal{F}_s^a$. From duality theory, the corresponding dual inner problem has $N + 1 + D_s^a$ unknowns and $2N + 1 + D_s^a$ constraints. Overall, Problem (9) has $O\left((N + 1 + D)MQ\right)$ more unknowns and $O\left((2N + 1 + D)MQ\right)$ more constraints of Problem (6), so its size is polynomial in $\mathcal{R}$. If $\mathcal{M}_\mathcal{C}$ satisfies Assumption 5.1, Problem (9) is convex. Using Proposition 4.1, we conclude that it can be solved in time polynomial in $\mathcal{R}$. Finally, when strong duality holds for the transformation in Equation (8), soundness and completeness of the final solution are preserved because the dual and primal optimal value of each inner problem are equivalent. □

6 Case Studies

We implemented the proposed verification algorithm in Python, and interfaced it with PRISM [5] to extract information about the CMDP model. We used MOSEK [25] to solve the LPs generated for the interval model and implemented customized numerical solvers for the other models of uncertainty. The implemented tool is available at [26]. The algorithm was tested on all the case studies collected in the PRISM benchmark suite [27]. Due to space limits, we report only two of them: the verification of a consensus protocol and of a dynamic configuration protocol for IPv4 addresses. The runtime data were obtained on a 2.4 GHz Intel Xeon with 32GB of RAM.

6.1 Consensus Protocol

Consensus problems arise in many distributed environments, where a group of distributed processes attempt to reach an agreement about a decision to take by accessing some shared entity. A consensus protocol ensures that the processes will eventually terminate and take the same decision, even if they start with initial guesses that might differ from one another.

We analyze the randomized consensus protocol presented in [19, 28]. The protocol guarantees that the processes return a preference value $v \in \{1, 2\}$, with probability parameterized by a process independent value R ($R \geq 2$) and the number of processes P. The processes communicate with one another by accessing a shared counter of value c. The protocol proceeds in rounds. At each round, a process flips a local coin, increments or decrements the shared counter depending on the outcome and then reads its value c. If $c \geq PR$ ($c \leq -PR$), it chooses $v = 1$ ($v = 2$). Note that the larger the value of R, the longer it takes on average for the processes to reach the decision. Nondeterminism is used to model the asynchronous access of the processes to the shared counter, so the overall protocol is modeled as an MDP.

We verify the property **Agreement:** all processes must agree on the same decision, i.e., choose a value $v \in \{1, 2\}$. We compute the minimum probability of **Agreement** and compare it against the theoretical lower bound $(R-1)/2R$ [19]. In PCTL syntax:

$$P_{s_0}^{min}[\psi] := P_{s_0}^{min}(\mathbf{F}(\{finished\} \wedge \{all\_coins\_equal\_1\})) \tag{12}$$

We consider the case where one of the processes is unreliable or adversarial, i.e., it throws a biased coin instead of a fair coin. Specifically, the probability of either outcome lies in the uncertainty interval $[(1-u)p_0, (1+u)p_0]$, where $p_0 = 0.5$ according to the protocol. This setting is relevant to analyze the protocol robustness when a process acts erroneously due to a failure or a security breach. In particular, our approach allows to study attacks that deliberately hide under the noise threshold of the protocol. In such attacks, the compromised node defers agreement by producing outputs whose statistical properties are within the noise tolerance of an uncompromised node, so that it is harder to detect its malicious behavior.

Figure 1 shows the effect of different levels of uncertainty on the computed probabilities for $P = 4$. With no uncertainty ($u = 0$), $P_{s_0}^{min}$ increases as R increases, because a larger R drives the decision regions further apart, making it more difficult for the processes to decide on different values of v. As R goes to infinity, $P_{s_0}^{min}$ approaches the theoretical lower bound $lim_{R\to\infty}(R-1)/2R = 0.5$. However, even with a small uncertainty ($u = 0.01$), $P_{s_0}^{min}$ soon decreases for increasing R. With a large uncertainty ($u = 0.15$), $P_{s_0}^{min}$ quickly goes to 0. A possible explanation is that the faulty process has more opportunities to deter agreement for a high R, since R also determines the expected time to termination. Results thus show that the protocol is vulnerable to uncertainties. This fact may have serious security implication, i.e., a denial-of-service attack could reduce the availability of the distributed service, since a compromised process may substantially alter the expected probability of agreement.

Lastly, we study the scalability of the CP procedure, by evaluating Equation (12) while sweeping R both for $P = 2$ and $P = 4$. We use MOSEK [25] to solve Problem (9) and set the Time Out (TO) to one hour. In Figure 2, we plot the sum $(N + T)$ of the number of states (N) and transitions (T) of the CMDP, which are independent of the uncertainty in the transition probabilities, to represent the model size (top), the sum $(V + C)$ of the number of variables (V) and constraints (C) of the generated LP instances of Problem (9) (center), and the running time t_{CP} (bottom). $V+C$ always scales linearly with $N + T$ (the lines have the same slope), supporting the polynomial complexity result for our algorithm. Instead, t_{CP} scales linearly only for smaller problems ($P = 2$), while it has a higher-order polynomial behavior for larger problems ($P = 4$) (the line is still a straight line but with steeper slope, so it is polynomial on logarithmic axes). This behavior depends on the performance of the chosen numerical solver, and it can improve by benefiting from future advancements in the solver implementation. In Table 3, we compare the CP procedure with two tools, PRISM [5] and PARAM [18], in terms of runtime, for varying values of P and R. Although neither tool solves the same problem addressed in this paper, the comparison is useful to assess the practicality of the proposed approach. In particular, PRISM only verifies PCTL properties of MDPs with no uncertainties. PARAM instead derives a symbolic expression of the satisfaction probabilities as a function of the model parameters, to then find the parameter

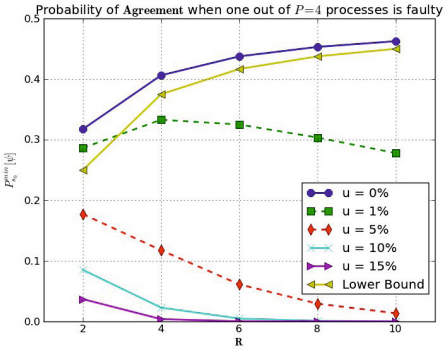


Fig. 1. Value of Eq. 12 in function of R while varying the uncertainty level u

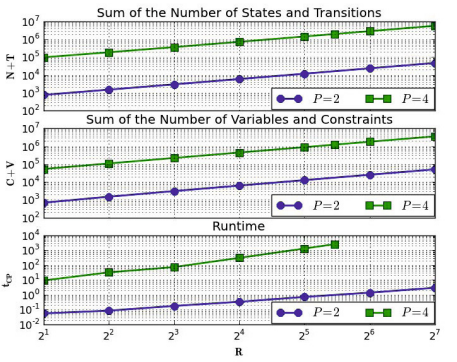


Fig. 2. Scalability of the CP procedure

values that satisfy the property. Hence, PRISM only considers a special case of the models considered in this paper, while our approach only returns the worst-case scenario computed by PARAM. Results show that the CP procedure runs faster than PRISM for some benchmarks, but it is slower for larger models. This is expected since the scalability of our approach depends mainly on the problem size, while the performance of the iterative engine in PRISM depends on the problem size and on the number of iterations required to achieve convergence, which is dependent on the problem data. Finally, our approach is orders of magnitude faster than PARAM, so it should be preferred to perform worst-case analysis of system performances.

6.2 ZeroConf Dynamic Configuration Protocol for IPv4 Link-Local Addresses

The ZeroConf protocol [29,30] is an Internet Protocol (IP)-based configuration protocol for local (e.g. domestic) networks. In such a local context, each device should configure its own unique IP address when it gets connected to the network, with no user intervention. The protocol thus offers a distributed "plug-and-play" solution in which address configuration is managed by individual devices when they are connected to the network. The network is composed of DV_{tot} devices. After being connected, a new device chooses randomly an IP address from a pool of $IP_A = 65024$ available ones, as specified by the standard. The address is non-utilized with probability $p_0 = 1 - DV_{tot}/IP_A$. It then sends messages to the other devices in the network, asking whether the chosen IP

Table 3. Runtime Comparison

Tool	$P=2, R=2$ $N+T=764$	$R=7$ $2,604$	$R=128$ $47,132$	$P=4, R=2$ $97,888$	$R=32$ $1,262,688$	$R=44$ $1,979,488$	$P=6, R=4$ $14,211,904$
CP	$0.02s$	$0.1s$	$2.1s$	$8.3s$	$1,341s$	$2,689$	TO
PRISM	$0.01s$	$0.09s$	$196s$	$1s$	$2,047s$	TO	$1860s$
PARAM	$22.8s$	$657s$	TO	TO	TO	TO	TO

address is already in use. If no reply is received, the device starts using the IP address, otherwise the process is repeated.

The protocol is both probabilistic and timed: probability is used in the randomized selection of an IP address and to model the eventuality of message loss; timing defines intervals that elapse between message retransmissions. In [30], the protocol has been modeled as an MDP using the digital clock semantic of time. In this semantic, time is discretized in a finite set of epochs which are mapped to a finite number of states in an MDP, indexed by the epoch variable t_e. To enhance the user experience and, in battery-powered devices, to save energy, it is important to guarantee that a newly-connected device manages to select a unique IP address within a given deadline *dl*. For numerical reasons, we study the maximum probability of *not* being able to select a valid address within *dl*. In PCTL syntax:

$$P_{s_0}^{max}\left[\psi\right] := P_{s_0}^{max}\left(\neg\{unique\_address\}\ \mathcal{U}\ \{t_e > dl\}\right) \tag{13}$$

We analyzed how network performances vary when there is uncertainty in estimating: 1) the probability of selecting an IP address, and; 2) the probability of message loss during transmission. The former may be biased in a faulty or malicious device. The latter is estimated from empirical data, so it is approximated. Further, the IMDP semantic of IDTMCs (Section 1), which allows a nature to select a different transition distribution at each execution step, properly models the time-varying characteristics of the transmission channel.

In Figure 3, we added uncertainty only to the probability of message loss using the likelihood model, which is suitable for empirically-estimated probabilities. Using classical results from statistics [11], we computed the value of parameter β from Set (2) corresponding to several confidence levels C_L in the measurements. In particular, $0 \leq C_L \leq 1$ and $C_L = 1 - cdf_{\chi_d^2}\left(2 * (\beta_{max} - \beta)\right)$, where $cdf_{\chi_d^2}$ is the cumulative density function of the Chi-squared distribution with d degrees of freedom ($d = 2$ here because there are two possible outcomes, message lost or received). Results show that the value of $P_{s_0}^{max}$ increases by up to $\sim 10\times$ for decreasing C_L, while classical model-checking would only report the value for $C_L = 1$, which roughly over-estimates network performance. The plot can be used by a designer to choose *dl* to make the protocol robust to varying channel conditions, or by a field engineer to assess when the collected measurements are enough to estimate network performances.

In Figure 4, we compose different models of uncertainty, i.e., we also add uncertainty in the probability of selecting the new IP address using the interval model. This probability thus lies in the interval $[(1-u)p_0, (1+u)p_0]$. We arbitrarily fixed $dl = 25$ and swept DV_{tot} in the range $[10 - 100]$, which covers most domestic applications, to study how network congestion affects the value of Equation 13. We studied four scenarios: the *ideal* scenario, returned by classical model-checking techniques; the *confident, normal, conservative* scenarios, where we added increasing uncertainty to model different knowledge levels of the network behavior, a situation that often arises during the different design phases, from conception to deployment. Results show that $P_{s_0}^{max}\left[\psi\right]$ gets up to $\sim\ 15\times$ higher than the ideal scenario, an information that designers can use to determine the most sensitive parameters of the system and to assess the impact of their modeling assumptions on the estimation of network performances.

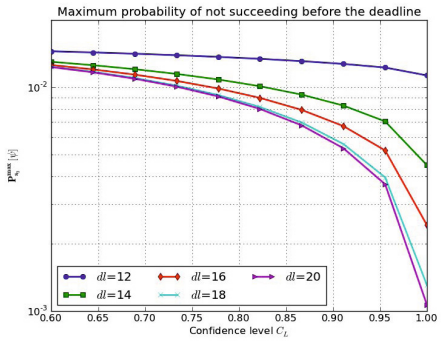


Fig. 3. Value of Equation 13 (top) and verification runtime (bottom)

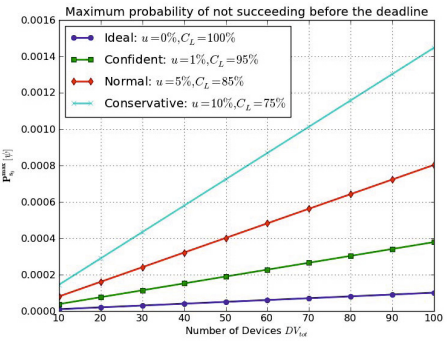


Fig. 4. Value of Eq. 13 for increasing number of devices in the network

7 Conclusions and Future Work

We addressed the problem of verifying PCTL properties of Convex-MDPs (CMDPs), i.e., MDPs whose transition probabilities lie within convex uncertainty sets. Using results on strong duality for convex programs, we proved that model checking is decidable in PTIME for the fragment of PCTL without the *Bounded Until* operator. For the entire PCTL syntax, the algorithmic complexity is pseudo-polynomial in the size of the property. Verification results on two case studies show that uncertainty can greatly alter the computed probabilities, thus revealing the importance of the proposed analysis.

As future work, we aim to relax the *rectangular uncertainty* assumption, to obtain a less conservative analysis. Also, we plan to verify a complex physical system, e.g. an airplane power system, in which modeling uncertainties are present both in the underlying physical process and in the failure probabilities of its components.

Acknowledgments. The authors thank John B. Finn for the contribution in the first stages of the project and the reviewers for their helpful comments. The research was partially funded by DARPA Award Number HR0011-12-2-0016 and by STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

1. Courcoubetis, C., Yannakakis, M.: The Complexity of Probabilistic Verification. Journal of ACM 42(4), 857–907 (1995)
2. Bianco, A., De Alfaro, L.: Model Checking of Probabilistic and Nondeterministic Systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
3. Kwiatkowska, M.: Quantitative Verification: Models, Techniques and Tools. In: Proc. of SIGSOFT, pp. 449–458 (2007)
4. Hansson, H., Jonsson, B.: A Logic for Reasoning About Time and Reliability. Formal Aspects of Computing 6(5), 512–535 (1994)
5. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of Probabilistic Real-Time Systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)

6. Kozine, I., Utkin, L.: Interval-Valued Finite Markov Chains. Reliable Computing 8(2), 97–113 (2002)
7. Sen, K., Viswanathan, M., Agha, G.: Model-Checking Markov Chains in the Presence of Uncertainties. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 394–410. Springer, Heidelberg (2006)
8. Chatterjee, K., Sen, K., Henzinger, T.: Model-Checking ω-regular Properties of Interval Markov Chains. In: Proc. of FOSSACS, pp. 302–317 (2008)
9. Ben-Tal, A., Nemirovski, A.: Robust Solutions of Uncertain Linear Programs. Oper. Res. Lett. 25(1), 1–13 (1999)
10. Andreychenko, A., Mikeev, L., Spieler, D., Wolf, V.: Parameter Identification for Markov Models of Biochemical Reactions. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 83–98. Springer, Heidelberg (2011)
11. Nilim, A., El Ghaoui, L.: Robust Control of Markov Decision Processes with Uncertain Transition Matrices. Journal of Operations Research, 780–798 (2005)
12. Puggelli, A., et al.: Polynomial-Time Verification of PCTL Properties of MDPs with Convex Uncertainties, UC-Berkeley, Tech. Rep. UCB/EECS-2013-24 (April 2013), http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-24.html
13. Vardi, M.Y.: Automatic Verification of Probabilistic Concurrent Finite State Programs. In: Proc. of SFCS, pp. 327–338 (1985)
14. Lehmann, E., Casella, G.: Theory of Point Estimation. Springer, New York (1998)
15. Puterman, M.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley and Sons (1994)
16. Forejt, V., Kwiatkowska, M., Norman, G., Parker, D.: Automated Verification Techniques for Probabilistic Systems. In: Bernardo, M., Issarny, V. (eds.) SFM 2011. LNCS, vol. 6659, pp. 53–113. Springer, Heidelberg (2011)
17. Barbuti, R., Levi, F., Milazzo, P., Scatena, G.: Probabilistic Model Checking of Biological Systems with Uncertain Kinetic Rates. In: Bournez, O., Potapov, I. (eds.) RP 2009. LNCS, vol. 5797, pp. 64–78. Springer, Heidelberg (2009)
18. Hahn, E.M., et al.: Synthesis for PCTL in Parametric Markov Decision Processes (2011)
19. Kwiatkowska, M., Norman, G., Segala, R.: Automated Verification of a Randomized Distributed Consensus Protocol Using Cadence SMV and PRISM. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 194. Springer, Heidelberg (2001)
20. Wolff, E., et al.: Robust Control of Uncertain Markov Decision Processes with Temporal Logic Specifications. In: CDC (2012)
21. D'Innocenzo, A., et al.: Robust PCTL Model Checking. In: Proc. of HSCC, pp. 275–286 (2012)
22. Clarke, E., Emerson, A.: Design and Synthesis of Synchronization Skeletons Using Branching Time Temporal Logic. In: Proc. of WLP, vol. 131 (1981)
23. Nesterov, Y., Nemirovski, A.: Interior-Point Polynomial Algorithms in Convex Programming. Studies in Applied and Numerical Mathematics (1994)
24. Boyd, S., Vandenberghe, L.: Convex Optimization. Cambridge University Press (2004)
25. MOSEK, http://www.mosek.com
26. http://www.eecs.berkeley.edu/~puggelli/
27. http://www.prismmodelchecker.org/benchmarks/
28. Aspnes, J., Herlihy, M.: Fast Randomized Consensus Using Shared Memory. Journal of Algorithms 11(3), 441–461 (1990)
29. Cheshire, S., Adoba, B., Gutterman, E.: Dynamic configuration of IPv4 link local addresses, http://www.ietf.org/rfc/rfc3927.txt
30. Kwiatkowska, M., et al.: Performance Analysis of Probabilistic Timed Automata Using Digital Clocks. Formal Methods in System Design 29, 33–78 (2006)

Faster Algorithms for Markov Decision Processes with Low Treewidth

Krishnendu Chatterjee[1] and Jakub Łącki[2]

[1] IST Austria (Institute of Science and Technology Austria)
[2] Institute of Informatics, University of Warsaw, Poland

Abstract. We consider two core algorithmic problems for probabilistic verification: the maximal end-component decomposition and the almost-sure reachability set computation for Markov decision processes (MDPs). For MDPs with treewidth k, we present two improved static algorithms for both the problems that run in time $O(n \cdot k^{2.38} \cdot 2^k)$ and $O(m \cdot \log n \cdot k)$, respectively, where n is the number of states and m is the number of edges, significantly improving the previous known $O(n \cdot k \cdot \sqrt{n \cdot k})$ bound for low treewidth. We also present decremental algorithms for both problems for MDPs with constant treewidth that run in amortized logarithmic time, which is a huge improvement over the previously known algorithms that require amortized linear time.

1 Introduction

In this work we will present efficient static and decremental algorithms for two core graph algorithmic problems in probabilistic verification when the graph has low treewidth. We start with the basic description of the model, the problem, and its importance.

Markov Decision Processes with Parity Objectives. The standard model of systems in probabilistic verification that exhibit both probabilistic and nondeterministic behavior are *Markov decision processes (MDPs)* [20]. MDPs have been used for control problems for stochastic systems [18], where nondeterminism represents the freedom of the controller to choose a control action, and the probabilistic component of the behavior describes the system response to control actions; as well as in many other applications [13,2,19]. A *specification* describes the set of good behaviors of the system. In the verification and control of stochastic systems the specification is typically an ω-regular set of paths. The class of ω-regular languages extends classical regular languages to infinite strings, and provides a robust specification language to express all commonly used specifications, such as safety, liveness, fairness, etc. [28]. A canonical way to define such ω-regular specifications are *parity* objectives. Hence MDPs with parity objectives provide the mathematical framework to study problems such as the verification and control of stochastic systems.

The Analysis Problems. There are two types of analysis for MDPs with parity objectives. The *qualitative analysis* problem given an MDP with a parity

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 543–558, 2013.

objective, asks for the computation of the set of states from where the parity objective can be ensured with probability 1 (almost-sure winning). The more general *quantitative analysis* asks for the computation of the maximal probability at each state with which the controller can satisfy the parity objective.

Significance of Qualitative Analysis. The qualitative analysis of MDPs is an important problem in verification. In several applications the controller must ensure that the correct behavior arises with probability 1. For example, in analysis of randomized embedded schedulers, the relevant questions is whether every thread progresses with probability 1 [15]. Moreover, even in applications where it is sufficient to satisfy the specification with probability $p < 1$, the correct choice of p is a challenging problem, due to the simplifications introduced during modeling; for example, for randomized distributed algorithms it is common to require correctness with probability 1 (see, e.g., [25,22,27]). Furthermore, in contrast to quantitative analysis, qualitative analysis is robust to numerical perturbations and precise transition probabilities, and consequently the algorithms for qualitative analysis are discrete and combinatorial. Finally, the best known algorithms for quantitative analysis of MDPs with parity objectives first perform the qualitative analysis, and then a quantitative analysis on the result of the qualitative analysis [13,14,10].

Core Algorithmic Problems. The qualitative analysis of MDPs with parity objectives relies on two graph algorithmic problems: (1) the maximal end-component decomposition; and (2) the almost-sure reachability set computation. An end-component C in an MDP is a set of states that is strongly connected and closed (no probabilistic transition from C leaves C), and a maximal end-component is an end-component which is maximal with respect to inclusion ordering. The maximal end-component (MEC) problem generalizes the scc (maximal strongly connected component) decomposition problem for directed graphs, and recurrent classes for Markov chains. The almost-sure reachability set for a set U of target vertices is the set of states such that it can be ensured that the set U is reached with probability 1 (in other words, it is the qualitative analysis for reachability objectives). The qualitative analysis problem for MDPs with parity objectives with d-priorities can be solved with $\log d$ calls to the MEC decomposition problem and one call to the almost-sure reachability problem [6]. Thus the MEC decomposition and the almost-sure reachability set computation are the core algorithmic problems required for the qualitative analysis of MDPs with parity objectives. In addition to qualitative analysis of MDPs with parity objectives, several algorithms for quantitative analysis of MDPs with quantitative objectives such as lim sup and lim inf objectives [8], combination of mean-payoff and parity objectives [9], and multi-objective mean-payoff objectives [5], rely crucially on the MEC decomposition problem.

Dynamic Algorithms. In the design and analysis of probabilistic systems it is natural that the systems under verification are developed incrementally by adding choices or removing choices for player 1, whereas the probabilistic choices which represent choice of nature or uncertainty remain unchanged. Hence there is a clear motivation to obtain dynamic algorithms for MEC decomposition and

almost-sure reachability set for MDPs that achieve a better running time than recomputation from scratch when player-1 edges are inserted or deleted.

Previous Results. The current best known algorithms for both the MEC decomposition and the almost-sure reachability set computation require $O(m \cdot \min(\sqrt{m}, n^{2/3}))$ time [6,7], where n is the number of states and m is the number of transitions (edges). Using a well-known fact that graphs of treewidth k have $O(n \cdot k)$ edges, one can obtain $O(n \cdot k \cdot \sqrt{n \cdot k})$ algorithms for MEC decomposition and almost-sure reachability set computation (they follow directly from the general $O(m \cdot \sqrt{m})$-time algorithm). The best known incremental and decremental algorithms for both problems require amortized linear time ($O(n)$ time) [6].

Our Contributions. In this work we consider MDPs with low treewidth. The concept of treewidth and tree decomposition of graphs was introduced in [26]. On one hand treewidth is a very relevant graph theoretic notion that measures how a graph can be decomposed into a tree, on the other hand, most systems developed in practice have low treewidth. For example, it has been shown that the control flow graphs of goto free Pascal programs have treewidth at most 3, and that the control flow graphs of goto free C programs have treewidth at most 6 [29]. It was also shown in [29] that tree decompositions, which are very costly to compute in general, can be generated in linear time with small constants for these control flow graphs. Our main results are efficient static and decremental algorithms for the MEC decomposition and the almost-sure reachability set computation for MDPs with low treewidth. Several benchmarks in PRISM are probabilistic programs written in programming languages mentioned above and consequently have small treewidth, and our results are relevant for such MDPs. The details of our contribution are as follows:

1. We present two improved static algorithms both for the MEC decomposition and the almost-sure reachability set computation for MDPs with treewidth k that run in time $O(n \cdot k^{2.38} \cdot 2^k)$ and $O(m \cdot \log n \cdot k)$, respectively, where n is the number of states and m is the number of edges (also note that for treewidth k we have $m = O(n \cdot k)$). For MDPs with low treewidth, our new linear-time algorithms are significant improvements over the previous known $O(n \cdot k \cdot \sqrt{n \cdot k})$ algorithms for both the problems.
2. We present decremental algorithms for the MEC decomposition and the almost-sure reachability set computation for MDPs with treewidth k that require $O(k \cdot \log n)$ amortized time, which is a huge improvement for constant treewidth over the previous algorithms that require $O(n)$ amortized time.

Our key technical contribution is as follows: for MDPs we establish a separation property for the almost-sure reachability set that allows us to use tree decomposition to obtain the $O(n \cdot k^{2.38} \cdot 2^k)$-time static algorithm. A similar intuition also works for the MEC decomposition problem. We then view the MEC decomposition and the almost-sure reachability set computation problems as decremental graph problems, and use dynamic graph algorithmic techniques to obtain the $O(m \cdot \log n \cdot k)$-time static algorithms and the decremental algorithms. Note that when edges are inserted, the treewidth of the graph may increase and the tree

decomposition can change. Thus, incremental algorithms with polylogarithmic amortized cost remain an interesting open question (even for scc decomposition). Proofs omitted for space available in [11].

Related Works. The notion of treewidth is studied in context of many graph theoretic algorithms, see [4] for an excellent survey. In verification, the problem of low and medium treewidth has been considered for efficient algorithms for parity games: a polynomial time algorithm for parity games with constant treewidth was presented in [24]; a recent improved result for constant treewidth was presented in [17]; and the algorithmic problem of parity games with medium treewidth was considered in [16]. Though the games problem has been studied with the treewidth restriction, to the best of our knowledge, improved algorithms for MDPs have not been considered with the treewidth restriction.

2 Preliminaries

In this section we first present the basic graph theoretic definitions of the MEC decomposition and the almost-sure reachability set computation, and then define the notions of treewidth.

2.1 MEC Decomposition and Almost-Sure Reachability

Markov decision processes (MDPs). A *Markov decision process (MDP)* $G = ((V, E), (V_1, V_P), \delta)$ consists of a finite directed *MDP graph* (V, E), a partition (V_1, V_P) of the *finite* set V of vertices, and a probabilistic transition function δ: $V_P \to \mathcal{D}(V)$, where $\mathcal{D}(V)$ denotes the set of probability distributions over the vertex set V, such that for all vertices $u \in V_P$ and $v \in V$ we have $uv \in E$ iff $\delta(u)(v) > 0$. An edge $uv \in E$ is a *player-1* edge if $u \in V_1$. For the algorithmic problems we will consider, the probabilistic transition function will not be relevant and we will consider the MDP graph along with the partition.

Maximal end-component decomposition. For the maximal end-component decomposition, the input is a directed graph $G = (V, E)$ and a partition (V_1, V_P) of its vertex set (i.e., the MDP graph and the partition). An end-component U is a set of vertices such that the subgraph induced by U is strongly connected and for each edge $uv \in E$, if $u \in U \cap V_P$ then $v \in U$. If U_1 and U_2 are two end-components and $U_1 \cap U_2 \neq \emptyset$, then $U_1 \cup U_2$ is also an end-component. The maximal end-component (MEC) decomposition consists of all the maximal end-components of V and all vertices of V that do not belong to any MEC.

Almost-sure reachability. For almost-sure reachability, the input is an MDP and a target set $U \subseteq V$ of vertices, and the goal is to compute the set A of vertices, such that player 1 can ensure that the set U is reached with probability 1. We first note that given the target set U, we can add a new vertex s as the new target vertex, and transform the set U such that all out-edges from vertices in U end up in s, and the vertex s has only a self-loop. Thus we will consider the

case when the target set is a single vertex s. We first reduce the computation of the almost-sure reachability set for a target vertex s to the following problem. The input is a directed graph $G = (V, E)$, a partition (V_1, V_P) of its vertex set (the MDP graph and the partition), and a target vertex $s \in V$. The goal is to compute a maximal (w.r.t inclusion) subset $Q \subseteq V$, such that the following two conditions are satisfied:

- for every $q \in Q$, there exists a path from q to s consisting only of vertices in Q (global condition), and
- for every $uv \in E$, if $u \in Q \cap V_P$, then $v \in Q$ (local condition).

First observe that if $Q_1 \subseteq V$ and $Q_2 \subseteq V$ both satisfy the global and the local conditions, then so does $Q_1 \cup Q_2$. It follows that there is a unique maximum set $A^* \subseteq V$ that satisfies both the global and the local conditions. The resulting set A^* is the almost-sure reachability set (in the following also called an *ASR set*). Let A be the almost-sure reachability set and A^* be the largest set that satisfies the two conditions (the global and the local conditions).

Lemma 1. *We have $A = A^*$.*

Since $A = A^*$ we consider the graph theoretic problem of computation of A^* (i.e., the largest set satisfying the global and the local conditions).

Notations. Let G be a directed graph. We denote its vertex and edge set by $V(G)$ and $E(G)$, respectively. By $G[S]$ we denote the subgraph of G induced on vertices belonging to S, whereas by $G \setminus S$ we denote the subgraph of G induced on $V(G) \setminus S$. A *separator* is a subset $S \subseteq V(G)$, such that $G \setminus S$ has more connected components than G (when all edges are treated as undirected).

2.2 Tree Decomposition of Graphs

We begin by introducing some definitions depicted in Fig. 1.

Definition 1. *Let $G = (V, E)$ be an undirected graph. A* tree decomposition *of G is a pair (B, T), where B is a family $B_1, \ldots, B_n$ of subsets of V (called bags) and T is a tree, whose nodes are sets B_i. The decomposition satisfies the following properties:*

1. *$\bigcup B_i = V$ (bags cover vertices).*
2. *For every $uv \in E$ there exists B_j, such that $u, v \in B_j$ (bags cover edges).*
3. *For every $v \in V$ the sets B_i containing v form a connected subtree of T.*

Definition 2. *The* width *of a tree decomposition (B, T) is equal to $\max_{B_i \in B} |B_i| - 1$. The treewidth of an undirected graph is the minimal possible width of its tree decomposition.*

The concept of treewidth grasps the *sparseness* of a graph. Treewidth of a tree is equal to 1, while cliques on n vertices have treewitdth $n - 1$. Note that the definitions are given for undirected graphs, but they can also be applied to directed graphs. In such case, we treat all edges as undirected.

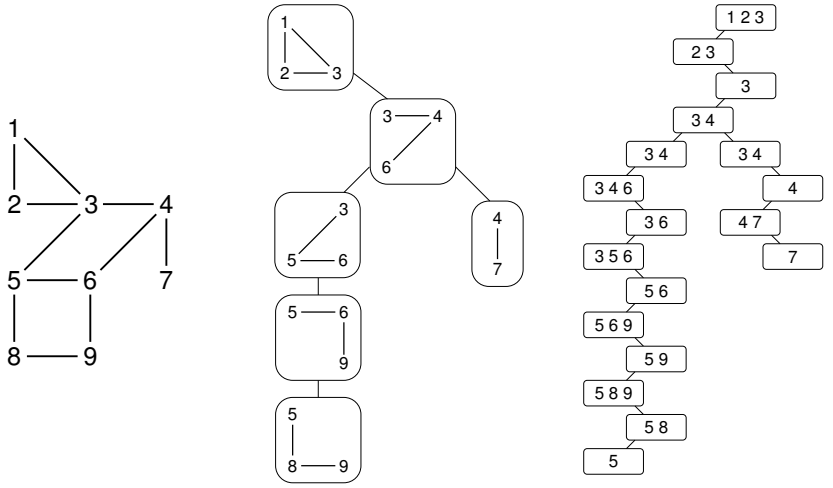


Fig. 1. A sample graph (left), its tree decomposition (center, edges covered by each bag have been marked for illustration) and a nice tree decomposition (right)

Definition 3. *A tree decomposition* (B, T) *is called* nice *if* T *is a rooted tree and each of its nodes* B_i *belongs to one of the following four types:*

1. ***leaf*** — B_i *is a leaf of* T *and* $|B_i| = 1$.
2. ***introduce*** — B_i *has a single child* B_j *and* $B_i = B_j \cup \{v\}$.
3. ***forget*** — B_i *has a single child* B_j *and* $B_i = B_j \setminus \{v\}$.
4. ***join*** — B_i *has two children* B_j *and* B_k*, and* $B_i = B_j = B_k$.

Theorem 1 ([3]). *Let* G *be a graph of treewidth* k*. Assuming that* k *is a constant, the tree decomposition of* G *of width* k *can be computed in* $O(n)$ *time.*

Lemma 2 (see e.g. [21]). *A tree decomposition can be transformed, in linear time, into a nice tree decomposition of the same width, consisting of* $O(n)$ *nodes.*

We also use the following well-known fact, which can be derived from the definition. Informally, consider a vertex t_B of a tree decomposition T of a graph G. Assume that it contains a bag $B \subset V(G)$. Denote the connected components of $T \setminus \{t_B\}$ by $T_1, \ldots, T_k$. Then, the trees T_i correspond to connected components of $G \setminus B$, namely bags from each T_i cover vertices from one connected component.

Lemma 3. *Let* B *be a bag in a node* t_B *of the tree decomposition of* G*. Consider the connected components* $T_1, \ldots, T_k$ *of* $T \setminus \{t_B\}$*. Then the following hold:*

1. *Either* B *is a separator in* G *or all but one* T_i *consist solely of bags that are subsets of* B*.*
2. *Each path from a vertex* $u \notin B$ *covered with a bag in* T_i *to a vertex* $v \notin B$ *covered with a bag in* T_j $(i \neq j)$ *goes through a vertex in* B*.*

Observe that a vertex not belonging to B can be covered by bags from at most one T_i. This is because the set of bags covering a given vertex forms a connected subgraph of T.

3 Algorithms for MDPs with Constant Tree-Width

In this section we will first present an algorithm for computing the ASR set, whose running time depends linearly on the size of the input MDP graph, where the input graph has constant treewidth. We will then present the linear-time algorithm for MEC decomposition for MDPs with constant treewidth graphs. The algorithms require that a tree decomposition of the graph of width k is given and run in time that is exponential in k. If k is a constant, the decomposition can be computed in linear time (see Theorem 1). To simplify presentation, we use Lemma 2 to transform the decomposition to a *nice* one.

3.1 Almost-Sure Reachability

Our algorithm for the ASR set computation is based on the following separation property.

Lemma 4. *Let B be a subset of $V(G)$, such that the target vertex s belongs to B. Denote the connected components of $G \setminus B$ by $C_1, \ldots, C_k$. Assume that we know the intersection of the ASR set A with B. For each $i = 1, \ldots, k$, construct the subgraph of G induced on $C_i \cup B$. Add to this graph a set of edges $\{vs | v \in A \cap B\}$, thus obtaining a* patched *component $\overline{C}_i$. Denote by A_i the ASR set in $\overline{C}_i$. Then we have $A = A_1 \cup \ldots \cup A_k$.*

Lemma 4 says that if we know $A \cap B$, then we can compute the ASR set independently in each (patched) connected component of $G \setminus B$ and then simply merge the results. Since we assume that G has low treewidth, it also has separators of small size. Thus, in the algorithm we can guess $A \cap B$, by checking all possibilities. We do not prove the separation property explicitly. Instead, we give the algorithm inspired with this property and then prove its correctness. The property will follow from Lemma 6. Let us now describe the details.

Denote the nice tree decomposition of G by T. We add the target vertex s to every bag of T. Note that this might increase the width of the decomposition by at most one, but the modified T is still a valid tree decomposition. However, T is no longer a nice decomposition, as the leaf nodes now contain two vertices. We fix this, by adding a child $\{s\}$ to every leaf. Then we choose an arbitrary leaf as the root.

The algorithm is based on a bottom-up dynamic programming on T. Fix a node d of T, and assume that it contains a bag B_d. Denote by G_d the subgraph of G induced on the vertices enclosed in the bags from the subtree rooted at d. By Lemma 3, B_d separates $G_d \setminus B_d$ from the rest of the graph.

Now, according to Lemma 4, for each subset $B' \subseteq B_d$ we should add edges $\{vs | v \in B'\}$ to G_d and compute the ASR set of the obtained graph. However, we do a slightly different thing: instead of adding edges, we just treat all vertices of B' as target vertices (note that this has the same effect as adding edges from vertices in B' to s). This motivates the following definition of a partial solution. Partial solution is defined with respect to a subgraph of $G_d \subseteq G$, and, informally, it is the set of vertices from G_d that will be included in the ASR set.

Definition 4. *A* partial solution *for a node d is a subset of $V(G_d)$. A partial solution P is called* valid*, if the following hold.*

i. For every $v \in P \cap V_P$ and every edge $vu \in E(G_d)$, we have $u \in P$.
ii. For every $v \in P$ there exists a path in P that connects v to some vertex in $P \cap B_d$.

We denote by $P(B', d)$ the maximal (w.r.t. inclusion) valid partial solution (for node d) which satisfies $P(B', d) \cap B_d = B'$.[1] Observe that the definition is unambiguous, since the union of two valid partial solutions is a valid partial solution. However, it might be the case that for some choice of B' there are no feasible valid partial solutions. In such a case we set $P(B', d) = \bot$. We later show that if $B' = A \cap B_d$, then $P(B', d) = A \cap V(G_d)$.

The algorithm considers possible ways of including a subset of B_d in the ASR set, by iterating through all *valid* subsets $B' \subseteq B_d$. A subset $B' \subseteq B_d$ is valid, if it contains the target s and for each $v \in B' \cap V_P$ and every edge $vu \in E \cap (B_d \times B_d)$, we have $u \in B'$. In particular, for any valid partial solution P containing s, the set $P \cap B_d$ is a valid subset.

In addition to $P(B', d)$, for each valid $B' \subseteq B$ and each pair of vertices $x, y \in B'$, we compute whether there exists an x-to-y path consisting of vertices contained in $P(B', d)$. Formally, we compute the transitive closure of $G[P(B', d)]$, restricted to B'. In the following this transitive closure is denoted by $TC(B', d)$. Note that it is a subset of $B_d \times B_d$.

The algorithm is run bottom-up on T. For a given node d and each valid subset B' it computes $P(B', d)$ and $TC(B', d)$, using the values from the children of d. There are four cases to consider, one for each type of node. In the description, we assume that the value $\bot$ is *propagating*. This means, that the result of any set operation involving $\bot$ is $\bot$.

- **Leaf** The bag contains a single vertex s (the target), so the transitive closure is empty and we set $P(\{s\}, d) = \{s\}$.
- **Join** Denote the children of d by c_1 and c_2. In this case, we set $P(B', d) = P(B', c_1) \cup P(B', c_2)$, so the transitive closures from the children have to be combined, i.e. $TC(B', d) = (TC(B', c_1) \cup TC(B', c_2))^*$. The asterisk denotes the operation of computing the transitive closure.
- **Introduce** Denote the introduced vertex by w and the child of d by c. For all valid subsets $B' \subseteq B_d$ that do not contain w, we set $P(B', d) = P(B', c)$ and $TC(B', d) = TC(B', c)$. If $w \in B'$, then $P(B', d) = P(B' \setminus \{w\}, c) \cup \{w\}$. Thus, to compute the transitive closure in this case, we take $TC(B' \setminus \{w\}, c)$, add all edges incident to w and compute the transitive closure of the obtained set. Hence, $TC(B', d) = (TC(B' \setminus \{w\}, c) \cup \{wz \in E(G) | z \in B'\} \cup \{zw \in E(G) | z \in B'\})^*$.
- **Forget** Denote the vertex that is forgotten by w and the child of d by c. Hence, the bag in the child B_c is equal to $B_d \cup \{w\}$. We check whether we

[1] In the end we prove slightly less about the values $P(\cdot, \cdot)$ that are computed by the algorithm, but it is convenient to think about them this way.

can include w in $P(B', d)$. For this, condition (ii) (of Definition 4) has to hold, i.e., there has to be a path in $P(B', d)$ that connects w to some vertex in B'. We claim that it suffices to check, whether w has any out-edges in $TC(B' \cup \{w\}, c)$. If this is the case, then w is connected to some vertex from B' in $P(B' \cup \{w\}, c)$, so $P(B', d) = P(B' \cup \{w\}, c)$ and we can set $TC(B', d) = TC(B' \cup \{w\}, c) \cap (B' \times B')$. Otherwise, we just copy the result from the child, that is set $P(B', d) = P(B', c)$ and $TC(B', d) = TC(B', c)$.

Finally, the ASR set computed by the algorithm is stored in $P(\{s\}, r)$. We now prove the correctness of the algorithm with the following two lemmas (proof of Lemma 5 in [11]).

Lemma 5. *For each node d and each valid subset $B' \subseteq B_d$, if $P(B', d) \neq \bot$, then $P(B', d)$ is a valid partial solution and $TC(B', d)$ is computed correctly.*

Lemma 6. *Let A be the maximum ASR set. For each node d, $P(A \cap B_d, d) = A \cap V(G_d)$.*

Proof. The proof proceeds by induction on the depth of the subtree rooted in d. First, it is easy to see that $A \cap B_d$ is a valid subset for d. Moreover, $A \cap V(G_d)$ is a valid partial solution for d. Let us check condition (ii) of Definition 4. For each $v \in A$ there exists an v-to-s path p in A. Denote by v_l the last vertex of p that lies inside $A \cap V(G_d)$. By Lemma 3, $v_l \in B_d$ and consequently also $v_l \in A \cap B_d$.

- **Leaf** $P(A \cap B_d, d) = P(\{s\}, d) = \{s\} = A \cap V(G_d)$.
- **Join** By induction hypothesis we have $P(A \cap B_{c_i}, c_i) = A \cap V(G_{c_i})$, for $i = 1, 2$. From the definition $P(A \cap B_d, d) = P(A \cap B_d, c_1) \cup P(A \cap B_d, c_2) = P(A \cap B_{c_1}, c_1) \cup P(A \cap B_{c_2}, c_2) = (A \cap V(G_{c_1})) \cup (A \cap V(G_{c_2})) = A \cap (V(G_{c_1}) \cup V(G_{c_2})) = A \cap V(G_d)$.
- **Introduce** If A does not contain the introduced vertex w, then $P(A \cap B_d, d) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap (V(G_d) \setminus \{w\}) = A \cap V(G_d)$. Otherwise, if $w \in A$ we have $P(A \cap B_d, d) = P((A \cap B_d) \setminus \{w\}, c) \cup \{w\} = (A \cap V(G_c)) \cup \{w\} = A \cap (V(G_d) \setminus \{w\}) \cup \{w\} = A \cap V(G_d)$.
- **Forget** Denote the forgotten vertex by w.
  We claim that $w \in A$ iff $A \cap B_c$ is a valid subset of B_c and w has some out-edges in $TC(A \cap B_c, c)$. ($\Rightarrow$) It follows immediately that $A \cap B_c$ is a valid subset. Moreover, since there is a path from w to s in A, by Lemma 3, there has to be a path that connects w to some vertex in $(A \cap B_c) \setminus \{w\}$ in $P(A \cap B_c, c)$. ($\Leftarrow$) Assume that $w \notin A$. We show that $A \cup P((A \cap B_c) \cup \{w\}, c)$ is an almost-sure reachable set that is larger than A. Indeed, we know that from every vertex in $P((A \cap B_c) \cup \{w\}, c)$ there is a path to a vertex in $(A \cap B_c) \cup \{w\}$, hence also a path to $A \cap B_c$. In addition, from every vertex in $A \cap B_c$ there is a path to s. It follows easily that condition (ii) of being an ASR set also holds, which shows the desired.
  Now, if $w \in A$, the algorithm sets $P(A \cap B_d, d) = P((A \cap B_d) \cup \{w\}, c) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap V(G_d)$. On the other hand, if $w \notin A$, we have $P(A \cap B_d, d) = P(A \cap B_d, c) = P(A \cap (B_c \setminus \{w\}), c) = P(A \cap B_c, c) = A \cap V(G_c) = A \cap V(G_d)$. □

By applying Lemma 6 to the root r of the tree decomposition, we obtain that $P(A \cap V(G), r) = A \cap V(G) = A$. Let us now analyze the running time.

Running time analysis. We represent $TC(\cdot, \cdot)$ with a $(k+2) \times (k+2)$ matrix. (In the original tree decomposition bags had size $k+1$, but then we added the vertex s to every bag.) The sets $P(\cdot, \cdot)$ can be represented implicitly, that is for a set $P(B, d)$ we store how it can be obtained from the respective sets contained in the children of d. This requires constant memory for each set. We iterate through $O(2^k)$ subsets of each bag. Checking whether a set is valid boils down to inspecting all edges inside a bag, which can be done in $O(k^2)$ time. The most costly operation performed for each valid subset is the computation of the transitive closure of a graph containing $O(k)$ vertices. This can be achieved in $O(k^{2.38})$ time by using fast matrix multiplication ([12], [30]).[2] Restoring the result takes time that is linear in the size of the tree decomposition. By Lemma 2, the decomposition consists of $O(n)$ nodes. Hence, the algorithm runs in $O(n \cdot 2^k \cdot k^{2.38})$ time. We obtain the following result.

Theorem 2. *Given an MDP and its tree decomposition of width k of the MDP graph, the ASR set can be computed in $O(n \cdot 2^k \cdot k^{2.38})$ time, where n is the number of states (vertices).*

3.2 MEC Decomposition

The algorithm is similar to the one for the ASR set in that it is also based on dynamic programming on a tree decomposition. Again, we assume that we have a nice tree decomposition with a bag of size 1 in the root. This time we obviously do not add the target vertex to every bag, as there is no distinguished vertex.

As in the previous algorithm, we define a partial solution for a node d to be a subset of $V(G_d)$. This subset consists of vertices that are to form a single MEC. A partial solution P is valid, if three conditions hold.

1. For every $v \in P \cap V_P$ and every edge $vu \in E(G_d)$, we have $u \in P$.
2. For every $v \in P$ there exists a path in P from v to some vertex in $P \cap B_d$.
3. For every $v \in P$ there exists a path in P from some vertex in $P \cap B_d$ to v.

Note that the only difference from the algorithm for ASR set is that we have added the third condition. As a result we can use the dynamic programming scheme from the previous section, with only a slight change. When we perform a check that depends on the second condition (while processing a **forget** node), we need to run two symmetric checks instead of one. Let $P(B', d)$ denote the maximal partial solution for d such that $P(B', d) \cap B_d = B'$.

We use the following two lemmas to show the correctness of the algorithm, and the proofs are analogous to lemmas in the previous section.

Lemma 7. *For each node d and each valid subset $B' \subseteq B_d$, $P(B', d)$ is a valid partial solution and $TC(B', d)$ is computed correctly.*

[2] In practice, a simple k^3 algorithm might a better choice than algebraic algorithms for multiplying matrices.

Lemma 8. *For every node d and MEC M such that $M \cap B_d \neq \emptyset$, we have $P(M \cap B_d, d) = M \cap V(G_d)$.*

The difference in this algorithm is in obtaining the result after the dynamic programming step is finished. First, we find the rootmost (that is, the one closest to the root) node d_1 and a vertex $v_1 \in B_{d_1}$, such that $P(\{v_1\}, d_1) \neq \bot$. In case of a tie, we can choose any node. We claim that $M_1 = P(\{v_1\}, d_1)$ is a MEC . We repeat this procedure, without taking into account vertices from M_1. This process is continued, as long as a feasible node and vertex can be found. We now show that it is correct.

Lemma 9. *For each node d and $v \in B_d$, if $P(\{v\}, d) \neq \bot$, then $P(\{v\}, d)$ is an end-component of G.*

Proof. From the definition of $P(\cdot, \cdot)$, we have that for every $u \in P(\{v\}, d) \cap V_P$ and every $ux \in E$, it holds that $x \in P(\{v\}, d)$. Moreover, from each vertex of $P(\{v\}, d)$ there is a path to v and from v there is a path to each vertex of $P(\{v\}, d)$. It follows that there is a path between any pair of vertices in $P(\{v\}, d)$, so it is a strongly connected set in G, thus also an end-component. □

This implies that our algorithm finds a collection of end-components. We now show that each such end-component is a MEC. Let M be an arbitrary MEC and let d be the rootmost node, such that $B_d \cap M \neq \emptyset$. Since the tree decomposition is nice, $B_d \cap M$ contains a single vertex v. From Lemma 8 it follows that $M = P(\{v\}, d)$. It is easy to see that when the algorithm picks a first vertex from M, it picks the vertex v defined above, and thus finds a MEC M. It follows easily that every MEC is eventually found by the algorithm.

Let us now discuss the running time. As before, the dynamic programming step requires $O(n \cdot 2^k \cdot k^{2.38})$ time. Retrieving all MECs from their implicit representations requires time that is bounded by the total time of building these representations. Moreover, the process of finding rootmost nodes requires time that is linear in the size of the tree decomposition. Hence, the running time is bounded by the time of the dynamic programming and amounts to $O(n \cdot 2^k \cdot k^{2.38})$.

Theorem 3. *Given an MDP and the tree decomposition of width k of the MDP graph, the MEC decomposition can be computed in $O(n \cdot 2^k \cdot k^{2.38})$ time, where n is the number of states (vertices).*

4 Static and Decremental Algorithms for MEC Decomposition and Almost-Sure Reachability

In this section we will present the $O(m \cdot k \cdot \log n)$-time static algorithms for the MEC decomposition and the ASR set computation, and the decremental algorithms. The key would be to present two simple algorithms for the problems that we will view as decremental graph algorithmic problems (decremental scc computation for MEC decomposition, and decremental directed reachability for

ASR computation). We will then use dynamic graph algorithmic techniques to obtain the desired result. We start with the two basic algorithms. The most straightforward implementations of both these algorithms are not efficient, but we later show that they can be speeded up significantly for graphs with low treewidth using dynamic graph algorithmic techniques.

4.1 Basic Algorithms

MEC Decomposition. We first give an algorithm (formal description as Algorithm 1) for computing MEC decomposition. Here, ComputeSccs denotes a function, which computes an array SCC that maps the vertices v into unique identifiers $SCC[v]$ of the strongly connected components in the graph.

Algorithm 1. Mec(G)

1: $G' := G$
2: $SCC :=$ ComputeSccs(G')
3: **while** $\exists_{u \in V_P \cap V(G')} \exists_{uv \in E(G)} SCC[u] \neq SCC[v]$ **do**
4: remove u from G'
5: $SCC :=$ ComputeSccs(G')

Lemma 10. *Algorithm 1 is correct.*

Proof. The algorithm removes a subset of vertices of G, thus obtaining a graph G'. It follows clearly that once the algorithm terminates, the strongly connected components of G' form a MEC decomposition of G'. Moreover, they are end-components in G (note that we use $E(G)$ instead of $E(G')$ in the condition in the third line). To show that these sets form a MEC decomposition for G (i.e., they are maximal with respect to inclusion), we prove that every vertex u that is removed does not belong to any MEC of G. If u belongs to some MEC M, then v must also belong to M. But, by the definition of a strongly connected component, u is not reachable from v, so they cannot belong to the same MEC. Hence, u is not contained in any MEC. □

Algorithm 2. Asr(G, s)

1: $G' := G$
2: $A :=$ FindReachable(G', s)
3: **while** $\exists_{u \in V_P \cap A} \exists_{uv \in E(G)} v \notin A$ **do**
4: remove u from G'
5: $A :=$ FindReachable(G', s)

Almost-Sure Reachability. A similar algorithm to the one above can be given for ASR. Procedure FindReachable computes the set of vertices that are connected to s with a path in G. The formal description is given as Algorithm 2.

Lemma 11. *Algorithm 2 is correct.*

Proof. The algorithm removes a subset of vertices of G, thus obtaining a graph G'. It follows clearly that once the algorithm terminates, the set of vertices from which there is a path to s is an ASR set in G' that satisfies both global and local conditions. To show that it is also an ASR set in G (i.e. it is maximal with respect to inclusion), we prove that every vertex u that is removed cannot belong to the ASR set. If u belonged to the set, then v would also belong to it. But there is no path from v to s in G, so v cannot belong to the ASR set, and neither can u. □

4.2 Static Algorithms for MEC and ASR

This section describes efficient implementations of algorithms from Section 4.1 that work for graphs with low treewidth.

MEC Decomposition. In order to compute MEC decomposition, we need to give an efficient implementation of Algorithm 1. This consists in maintaining the array SCC under a sequence of vertex deletions. Note that instead of removing vertices, we might well just remove all its incident edges.

To maintain strongly connected components we use a data structure by Łącki [23]. Given the tree decomposition of a graph of width k, it can maintain the SCC array subject to edge deletions. The total running time of all delete operations is $O(m \cdot k \cdot \log n)$, and every query to the array is answered in constant time. Thus, if $\Omega(m)$ edges are deleted, the amortized time of one update is $O(k \cdot \log n)$.

After each update, if a strongly connected component decomposes into multiple strongly connected components, some edges that used to be contained in a single strongly connected component now connect different strongly connected components. It is easy to see that it suffices to check the condition from the third line of the algorithm just for these edges. The algorithm maintaining strongly connected components can be easily extended to report the desired edges with no additional overhead. This way, we obtain an algorithm that computes the MEC decomposition in $O(m \cdot k \cdot \log n)$ total time.

Almost-Sure Reachability. We now describe an efficient implementation of Algorithm 2. This time it suffices to give an efficient algorithm that maintains the subset $A \subseteq V$ of vertices, such that for every $r \in A$ there exists an r-to-s path in G. After reversing all edges in the graph this becomes a single-source reachability problem. We show that by modifying the algorithm of Łącki [23], this can be achieved in $O(k \cdot \log n)$ amortized time. We describe the details of the algorithm below.

Decremental single-source reachability. Given a directed graph G with a designated *source* $s \in V(G)$, the goal is to maintain the set of vertices reachable from s when the edges of G are deleted. Moreover, we assume that we are given the tree decomposition of G of width k.

The algorithm is a simplified version of the algorithm for decremental all-pairs reachability by Łącki [23]. The description in [23] contains an error in the running

time analysis of the all-pairs reachability. However, the problem disappears, if there is only a single source.

One of the ingredients of the algorithm is an algorithm for decremental single-source reachability in a DAG. The algorithm is very simple. In the beginning we delete all vertices that are not reachable from the source. Then, after an edge is deleted, we delete vertices (different from s) whose in-degree is 0, until all remaining vertices have positive in-degree. Note that deleting a vertex might decrease the in-degree of other vertices and trigger further deletions. The correctness of the algorithm follows easily. Moreover, it can be implemented, so that the total running time is linear in the number of edges of the initial graph. This is because every edge is examined when its start vertex is deleted and this means that the edge itself also gets deleted.

We can now proceed to the algorithm dealing with the general case. It maintains the subgraph of the initial graph that is reachable from s. In the description we treat G as a variable denoting this subgraph. To represent G we store its *condensation* G_c, that is the graph obtained from G by contracting all strongly connected components. It is easy to see that a condensation of an arbitrary graph is acyclic. Hence, we can use the algorithm given above to maintain it. On the other hand, to maintain the strongly connected components of G, we use the data structure by Łącki [23].

When an edge belonging to the condensation is deleted, we can simply update the condensation DAG, deleting some vertices, if necessary. All other edges are contained inside strongly connected components, so the deletion is handled by the data structure. This might cause some strongly connected component to break. In such case the data structure can report the condensation of the subgraph obtained from breaking the component with no additional overhead. This subgraph is then planted in place of the appropriate vertex in the condensation. The details are given in [23].

The total running time of processing all edge deletions is $O(m \cdot k \cdot \log n)$ and the set of reachable vertices is maintained explicitly. Also recall that for treewidth k we have $m = O(n \cdot k)$.

Theorem 4. *Given an MDP and its tree decomposition of width k, the MEC decomposition and the ASR set can be computed in time $O(m \cdot k \cdot \log n)$, where n is the number of states (vertices) and m is the number of edges.*

4.3 Decremental Algorithms

Both algorithms that we have described can be easily extended to decremental algorithms that support edge deletions. However, only deleting edges $uv \in E$ such that $u \in V_1$ is allowed. This assures that the ASR set can only shrink and that every end-component in the MEC decomposition is a subset of a some end-component from the graph before the deletion.

Almost-Sure Reachability. The algorithm first runs Algorithm 2 during the initialization phase and computes the initial set A. The set A is maintained by a single-source decremental reachability algorithm. The very same high-level

algorithm can be used to update the set A after an edge is deleted. We run this algorithm whenever an edge is deleted. Observe that if we detect that A shrinks, i.e. a subset $U \subseteq A$ of vertices is removed from A, we need to check the condition in the third line only for edges that are entering this set. Thus, each edge is inspected at most once during the entire course of the algorithm. Hence, the dominating operation is the running time of the decremental single-source reachability algorithm, which requires $O(m \cdot k \cdot \log n)$ time over all deletions or $O(k \cdot \log n)$ amortized time for a single deletion, if $\Omega(m)$ edges are deleted. The proof of correctness is analogous to the one in Lemma 11.

MEC Decomposition. We use the same idea as for the decremental algorithm for the ASR set. In this case Algorithm 1 can be used both for the initialization and after an edge is deleted. By maintaining the array SCC with a data structure for decremental SCC maintenance, we get that the amortized time of processing a single update is $O(k \cdot \log n)$.

Theorem 5. *Given an MDP and its tree decomposition of width k, the MEC decomposition and the ASR set can be computed under the deletion of $\Omega(m)$ player-1 edges, in amortized time $O(k \cdot \log n)$ per edge deletion, where n is the number of states (vertices) and m is the number of edges.*

Concluding remarks. In this work, we presented faster static and decremental algorithms for two core algorithmic problems for MDPs when the treewidth is low. An interesting question for future work is whether the algorithms can be extended to MDPs with low DAG-width (as done for parity games in [1]).

Acknowledgements. The authors would like to thank Monika Henzinger for several interesting discussions on related topics. The research was supported by FWF Grant No P 23499-N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award. Jakub Łącki is a recipient of the Google Europe Fellowship in Graph Algorithms, and this research is supported in part by this Google Fellowship.

References

1. Berwanger, D., Dawar, A., Hunter, P., Kreutzer, S., Obdrzálek, J.: The DAG-width of directed graphs. J. Comb. Theory, Ser. B 102(4), 900–923 (2012)
2. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In: Thiagarajan, P.S. (ed.) FSTTCS 1995. LNCS, vol. 1026, pp. 499–513. Springer, Heidelberg (1995)
3. Bodlaender, H.L.: A linear-time algorithm for finding tree-decompositions of small treewidth. SIAM J. Comput. 25(6), 1305–1317 (1996)
4. Bodlaender, H.L.: Treewidth: Algorithmic techniques and results. In: Privara, I., Ružička, P. (eds.) MFCS 1997. LNCS, vol. 1295, pp. 19–36. Springer, Heidelberg (1997)
5. Brázdil, T., Brozek, V., Chatterjee, K., Forejt, V., Kucera, A.: Two views on multiple mean-payoff objectives in Markov decision processes. In: LICS, pp. 33–42 (2011)
6. Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: SODA, pp. 1318–1336 (2011)

7. Chatterjee, K., Henzinger, M.: An $O(n^2)$ time algorithm for alternating büchi games. In: SODA, pp. 1386–1399 (2012)
8. Chatterjee, K., Henzinger, T.A.: Probabilistic systems with limsup and liminf objectives. In: Archibald, M., Brattka, V., Goranko, V., Löwe, B. (eds.) ILC 2007. LNCS, vol. 5489, pp. 32–45. Springer, Heidelberg (2009)
9. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 380–395. Springer, Heidelberg (2010)
10. Chatterjee, K., Jurdziński, M., Henzinger, T.: Quantitative stochastic parity games. In: SODA 2004, pp. 121–130. SIAM (2004)
11. Chatterjee, K., Łącki, J.: Faster algorithms for Markov decision processes with low treewidth CoRR abs/1304.0084 (2013), http://arxiv.org/abs/1304.0084
12. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. J. Symb. Comput. 9(3), 251–280 (1990)
13. Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. Journal of the ACM 42(4), 857–907 (1995)
14. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University (1997)
15. de Alfaro, L., Faella, M., Majumdar, R., Raman, V.: Code-aware resource management. In: EMSOFT 2005. ACM (2005)
16. Fearnley, J., Lachish, O.: Parity games on graphs with medium tree-width. In: Murlak, F., Sankowski, P. (eds.) MFCS 2011. LNCS, vol. 6907, pp. 303–314. Springer, Heidelberg (2011)
17. Fearnley, J., Schewe, S.: Time and parallelizability results for parity games with bounded treewidth. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 189–200. Springer, Heidelberg (2012)
18. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer (1997)
19. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 441–444. Springer, Heidelberg (2006)
20. Howard, H.: Dynamic Programming and Markov Processes. MIT Press (1960)
21. Kloks, T.: Treewidth, Computations and Approximations. LNCS, vol. 842. Springer (1994)
22. Kwiatkowska, M., Norman, G., Parker, D.: Verifying randomized distributed algorithms with prism. In: WAVE 2000 (2000)
23. Łącki, J.: Improved deterministic algorithms for decremental transitive closure and strongly connected components. In: SODA, pp. 1438–1445 (2011)
24. Obdržálek, J.: Fast mu-calculus model checking when tree-width is bounded. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 80–92. Springer, Heidelberg (2003)
25. Pogosyants, A., Segala, R., Lynch, N.: Verification of the randomized consensus algorithm of Aspnes and Herlihy: a case study. Dist. Comp. 13(3), 155–186 (2000)
26. Robertson, N., Seymour, P.D.: Graph minors. iii. planar tree-width. J. Comb. Theory, Ser. B 36(1), 49–64 (1984)
27. Stoelinga, M.: Fun with FireWire: Experiments with verifying the IEEE1394 root contention protocol. In: Formal Aspects of Computing (2002)
28. Thomas, W.: Languages, automata, and logic. In: Handbook of Formal Languages, vol. 3, ch. 7, pp. 389–455. Springer, Heidelberg (1997)
29. Thorup, M.: All structured programs have small tree-width and good register allocation. Inf. Comput. 142(2), 159–181 (1998)
30. Williams, V.V.: Multiplying matrices faster than coppersmith-winograd. In: STOC, pp. 887–898 (2012)

Automata with Generalized Rabin Pairs for Probabilistic Model Checking and LTL Synthesis

Krishnendu Chatterjee[1,⋆], Andreas Gaiser[2,⋆⋆], and Jan Křetínský[2,3,⋆⋆⋆]

[1] IST Austria
[2] Fakultät für Informatik, Technische Universität München, Germany
[3] Faculty of Informatics, Masaryk University, Brno, Czech Republic

Abstract. The model-checking problem for probabilistic systems crucially relies on the translation of LTL to deterministic Rabin automata (DRW). Our recent Safraless translation [KE12, GKE12] for the LTL(**F**,**G**) fragment produces smaller automata as compared to the traditional approach. In this work, instead of DRW we consider deterministic automata with acceptance condition given as disjunction of generalized Rabin pairs (DGRW). The Safraless translation of LTL(**F**,**G**) formulas to DGRW results in smaller automata as compared to DRW. We present algorithms for probabilistic model-checking as well as game solving for DGRW conditions. Our new algorithms lead to improvement both in terms of theoretical bounds as well as practical evaluation. We compare PRISM with and without our new translation, and show that the new translation leads to significant improvements.

1 Introduction

Logic for ω-regular properties. The class of ω-regular languages generalizes regular languages to infinite strings and provides a robust specification language to express all properties used in verification and synthesis. The most convenient way to describe specifications is through logic, as logics provide a concise and intuitive formalism to express properties with very precise semantics. The linear-time temporal logic (LTL) [Pnu77] is the de-facto logic to express linear time ω-regular properties in verification and synthesis.

Deterministic ω-automata. For model-checking purposes, LTL formulas can be converted to nondeterministic Büchi automata (NBW) [VW86], and then the problem reduces to checking emptiness of the intersection of two NBWs (representing the system and the negation of the specification, respectively). However, for two very important problems deterministic automata are used, namely,

⋆ The author is supported by Austrian Science Fund (FWF) Grant No P 23499-N23, FWF NFN Grant No S11407-N23 (RiSE), ERC Start grant (279307: Graph Games), and Microsoft faculty fellows award.

⋆⋆ The author is supported by the DFG Graduiertenkolleg 1480 (PUMA).

⋆⋆⋆ The author is supported by the Czech Science Foundation, project No. P202/10/1469.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 559–575, 2013.

(1) the synthesis problem [Chu62, PR89]; and (2) the model-checking problem for probabilistic systems or Markov decision processes (MDPs) [BK08] which has a wide range of applications from randomized communication, to security protocols, to biological systems. The standard approach is to translate LTL to NBW [VW86], and then convert the NBW to a deterministic automata with Rabin acceptance condition (DRW) using Safra's determinization procedure [Saf88] (or using a recent improvement of Piterman [Pit06]).

Avoiding Safra's construction. The key bottleneck of the standard approach in practice is Safra's determinization procedure which is difficult to implement due to the complicated state space and data structures associated with the construction [Kup12]. As a consequence several alternative approaches have been proposed, and the most prominent ones are as follows. The first approach is the Safraless approach. One can reduce the synthesis problem to emptiness of nondeterministic Büchi tree automata [KV05]; it has been implemented with considerable success in [JB06]. For probabilistic model checking other constructions can be also used, however, all of them are exponential [Var85, CY95]. The second approach is to use heuristic to improve Safra's determinization procedure [KB06, KB07] which has led to the tool ltl2dstar [Kle]. The third approach is to consider fragments of LTL. In [AT04] several simple fragments of LTL were proposed that allow much simpler (single exponential as compared to the general double exponential) translations to deterministic automata. The generalized reactivity(1) fragment of LTL (called GR(1)) was introduced in [PPS06] and a cubic time symbolic representation of an equivalent automaton was presented. The approach has been implemented in the ANZU tool [JGWB07]. Recently, the $(\mathbf{F}, \mathbf{G})$-fragment of LTL, that uses boolean operations and only $\mathbf{F}$ (eventually or in future) and $\mathbf{G}$ (always or globally) as temporal operators, was considered and a simple and direct translation to deterministic Rabin automata (DRW) was presented [KE12]. Not only it covers all fragments of [AT04], but it can also express all complex fairness constraints, which are widely used in verification.

Probabilistic model-checking. Despite several approaches to avoid Safra's determinization, for probabilistic model-checking the deterministic automata are still necessary. Since probabilistic model-checkers handle linear arithmetic, they do not benefit from the symbolic methods of [PPS06, MS08] or from the tree automata approach. The approach for probabilistic model-checking has been to explicitly construct a DRW from the LTL formula. The most prominent probabilistic model-checker PRISM [KNP11] implements the ltl2dstar approach.

Our results. In this work, we focus on the $(\mathbf{F}, \mathbf{G})$-fragment of LTL. Instead of the traditional approach of translation to DRW we propose a translation to deterministic automata with *generalized Rabin pairs*. We present probabilistic model-checking as well as symbolic game solving algorithms for the new class of conditions which lead to both theoretical as well as significant practical improvements. The details of our contributions are as follows.

1. A Rabin pair consists of the conjunction of a Büchi (always eventually) and a coBüchi (eventually always) condition, and a Rabin condition is a disjunction of Rabin pairs. A generalized Rabin pair is the conjunction of conjunctions

of Büchi conditions and conjunctions of coBüchi conditions. However, as conjunctions of coBüchi conditions is again a coBüchi condition, a generalized Rabin pair is the conjunction of a coBüchi condition and conjunction of Büchi conditions.[1] We consider deterministic automata where the acceptance condition is a disjunction of generalized Rabin pairs (and call them DGRW). The $(\mathbf{F}, \mathbf{G})$-fragment of LTL admits a direct and algorithmically simple translation to DGRW [KE12] and we consider DGRW for probabilistic model-checking and synthesis. The direct translation of LTL($\mathbf{F}$,$\mathbf{G}$) could be done to a compact deterministic automaton with a Muller condition, however, the explicit representation of the Muller condition is typically huge and not algorithmically efficient, and thus reduction to deterministic Rabin automata was performed (with a blow-up) since Rabin conditions admit efficient algorithmic analysis. We show that DGRW allow both for a very compact translation of the $(\mathbf{F}, \mathbf{G})$-fragment of LTL as well as efficient algorithmic analysis. The direct translation of LTL($\mathbf{F}$,$\mathbf{G}$) to DGRW has the same number of states as for a general Muller condition. For many formulae expressing e.g. fairness-like conditions the translation to DGRW is significantly more compact than the previous ltl2dstar approach. For example, for a conjunction of three strong fairness constraints, ltl2dstar produces a DRW with more than a million states, translation to DRW via DGRW requires 469 states, and the corresponding DGRW has only 64 states.

2. One approach for probabilistic model-checking and synthesis for DGRW would be to first convert them to DRW, and then use the standard algorithms. Instead we present direct algorithms for DGRW that avoids the translation to DRW both for probabilistic model-checking and game solving. The direct algorithms lead to both theoretical and practical improvements. For example, consider the disjunctions of k generalized Rabin pairs such that in each pair there is a conjunction of a coBüchi condition and conjunctions of j Büchi conditions. Our direct algorithms for probabilistic model-checking as well as game solving is more efficient by a multiplicative factor of j^k and j^{k^2+k} as compared to the approach of translation to DRW for probabilistic model checking and game solving, respectively. Moreover, we also present symbolic algorithms for game solving for DGRW conditions.
3. We have implemented our approach for probabilistic model checking in PRISM, and the experimental results show that as compared to the existing implementation of PRISM with ltl2dstar our approach results in improvement of order of magnitude. Moreover, the results for games confirm that the speed up is even greater than for probabilistic model checking.

[1] Note that our condition (disjunction of generalized Rabin pairs) is very different from both generalized Rabin conditions (conjunction of Rabin conditions) and the generalized Rabin(1) condition of [Ehl11], which considers a set of assumptions and guarantees where each assumption and guarantee consists of *one* Rabin pair. Syntactically, disjunction of generalized Rabin pairs condition is $\bigvee_i(\mathbf{FG}a_i \wedge \bigwedge_j \mathbf{GF}b_{ij})$, whereas generalized Rabin condition is $\bigwedge_j(\bigvee_i(\mathbf{FG}a_{ij} \wedge \mathbf{GF}b_{ij}))$, and generalized Rabin(1) condition is $(\bigwedge_i(\mathbf{FG}a_i \wedge \mathbf{GF}b_i) \Rightarrow \bigwedge_j(\mathbf{FG}a_j \wedge \mathbf{GF}b_j))$.

2 Preliminaries

In this section, we recall the notion of linear temporal logic (LTL) and illustrate the recent translation of its (**F**,**G**)-fragment to DRW [KE12, GKE12] through the intermediate formalism of DGRW. Finally, we define an index that is important for characterizing the savings the new formalism of DGRW brings as shown in the subsequent sections.

2.1 Linear Temporal Logic

We start by recalling the fragment of linear temporal logic with *future* (**F**) and *globally* (**G**) modalities.

Definition 1 (LTL(F,G) syntax). *The formulae of the* (**F**,**G**)-fragment of linear temporal logic *are given by the following syntax:*

$$\varphi ::= a \mid \neg a \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathbf{F}\varphi \mid \mathbf{G}\varphi$$

where a ranges over a finite fixed set Ap of atomic propositions.

We use the standard abbreviations $\mathbf{tt} := a \vee \neg a$ and $\mathbf{ff} := a \wedge \neg a$. Note that we use the negation normal form, as negations can be pushed inside to atomic propositions due to the equivalence of $\mathbf{F}\varphi$ and $\neg\mathbf{G}\neg\varphi$.

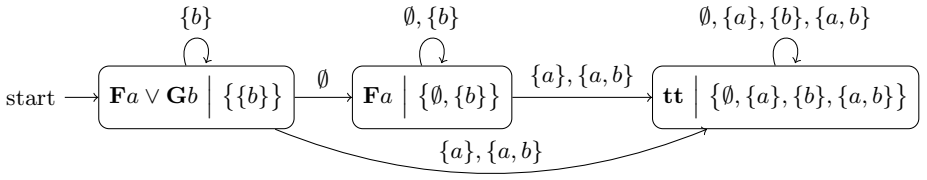
Definition 2 (LTL(F,G) semantics). *Let $w \in (2^{Ap})^\omega$ be a word. The ith letter of w is denoted $w[i]$, i.e. $w = w[0]w[1]\cdots$. Further, we define the ith suffix of w as $w_i = w[i]w[i+1]\cdots$. The semantics of a formula on w is then defined inductively as follows: $w \models a \iff a \in w[0]$; $w \models \neg a \iff a \notin w[0]$; $w \models \varphi \wedge \psi \iff w \models \varphi$ and $w \models \psi$; $w \models \varphi \vee \psi \iff w \models \varphi$ or $w \models \psi$; and*

$$\begin{aligned} w \models \mathbf{F}\varphi &\iff \exists\, k \in \mathbb{N}_0 : w_k \models \varphi \\ w \models \mathbf{G}\varphi &\iff \forall\, k \in \mathbb{N}_0 : w_k \models \varphi \end{aligned}$$

2.2 Translating LTL(F,G) into Deterministic ω-automata

Recently, in [KE12, GKE12], a new translation of LTL(**F**,**G**) to deterministic automata has been proposed. This construction avoids Safra's determinization and makes direct use of the structure of the formula. We illustrate the construction in the following examples.

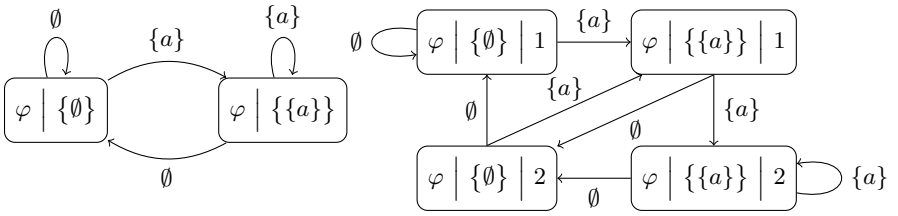
Example 3. Consider a formula $\mathbf{F}a \vee \mathbf{G}b$. The construction results in the following automaton. The state space of the automaton has two components. The first component stores the current formula to be satisfied. Whenever a letter is read, the formula is updated accordingly. For example, when reading a letter with no b, the option to satisfy the formula due to satisfaction of $\mathbf{G}b$ is lost and is thus reflected in changing the current formula to $\mathbf{F}a$ only.



The second component stores the last letter read (actually, an equivalence class thereof). The purpose of this component is explained in the next example. For formulae with no mutual nesting of **F** and **G** this component is redundant.

The formula $\mathbf{F}a \vee \mathbf{G}b$ is satisfied either due to $\mathbf{F}a$ or $\mathbf{G}b$. Therefore, when viewed as a Rabin automaton, there are two Rabin pairs. One forcing infinitely many visits of the third state (a in $\mathbf{F}a$ must be eventually satisfied) and the other prohibiting infinitely many visits of the second and third states (b in $\mathbf{G}b$ must never be violated). The acceptance condition is a disjunction of these pairs.

Example 4. Consider now the formula $\varphi = \mathbf{GF}a \wedge \mathbf{GF}\neg a$. Satisfaction of this formula does not depend on any finite prefix of the word and reading $\{a\}$ or $\emptyset$ does not change the first component of the state. This infinitary behaviour requires the state space to record which letters have been seen infinitely often and the acceptance condition to deal with that. In this case, satisfaction requires visiting the second state infinitely often *and* visiting the first state infinitely often.



However, such a conjunction cannot be written as a Rabin condition. In order to get a Rabin automaton, we would duplicate the state space. In the first copy, we wait for reading $\{a\}$. Once this happens we move to the second copy, where we wait for reading $\emptyset$. Once we succeed we move back to the first copy and start again. This bigger automaton now allows for a Rabin condition. Indeed, it is sufficient to infinitely often visit the "successful" state of the last copy as this forces infinite visits of "successful" states of all copies.

In order to obtain a DRW from an LTL formula, [KE12, GKE12] first constructs an automaton similar to DGRW (like the one on the left) and then the state space is blown-up and a DRW (like the one on the right) is obtained. However, we shall argue that this blow-up is unnecessary for application in probabilistic model checking and in synthesis. This will result in much more efficient algorithms for complex formulae. In order to avoid the blow-up we define and use DGRW, an automaton with more complex acceptance condition, yet as we show algorithmically easy to work with and efficient as opposed to e.g. the general Muller condition.

2.3 Automata with Generalized Rabin Pairs

In the previous example, the cause of the blow-up was the conjunction of Rabin conditions. In [KE12], a generalized version of Rabin condition is defined that allows for capturing conjunction. It is defined as a positive Boolean combination of Rabin pairs. Whether a set $\mathrm{Inf}(\rho)$ of states visited infinitely often on a run ρ is accepting or not is then defined inductively as follows:

$$\begin{aligned}
\mathrm{Inf}(\rho) \models \varphi \wedge \psi &\iff \mathrm{Inf}(\rho) \models \varphi \text{ and } \mathrm{Inf}(\rho) \models \psi \\
\mathrm{Inf}(\rho) \models \varphi \vee \psi &\iff \mathrm{Inf}(\rho) \models \varphi \text{ or } \mathrm{Inf}(\rho) \models \psi \\
\mathrm{Inf}(\rho) \models (F, I) &\iff F \cap \mathrm{Inf}(\rho) = \emptyset \text{ and } I \cap \mathrm{Inf}(\rho) \neq \emptyset
\end{aligned}$$

Denoting Q as the set of all states, (F, I) is then equivalent to $(F, Q) \wedge (\emptyset, I)$. Further, $(F_1, Q) \wedge (F_2, Q)$ is equivalent to $(F_1 \cup F_2, Q)$. Therefore, one can transform any such condition into a disjunctive normal form and obtain a condition of the following form:

$$\bigvee_{i=1}^{k} \left(\left(F_i, Q\right) \wedge \bigwedge_{j=1}^{\ell_i} \left(\emptyset, I_i^j\right) \right) \qquad (*)$$

Therefore, in this paper we define the following new class of ω-automata:

Definition 5 (DGRW). *An* automaton with generalized Rabin pairs (DGRW) *is a (deterministic) ω-automaton $\mathcal{A} = (Q, q_0, \delta)$ over an alphabet Σ, where Q is a set of states, q_0 is the initial state, $\delta : Q \times \Sigma \to Q$ is a transition function, together with a* generalized Rabin pairs (GRP) acceptance condition $\mathcal{GR} \subseteq 2^{2^Q \times 2^{2^Q}}$. *A run ρ of $\mathcal{A}$ is accepting for $\mathcal{GR} = \left\{ \left(F_i, \{I_i^1, \ldots, I_i^{\ell_i}\}\right) \;\middle|\; i \in \{1, \ldots, k\} \right\}$ if there is $i \in \{1, \ldots, k\}$ such that*

$$\begin{aligned}
&F_i \cap \mathrm{Inf}(\rho) = \emptyset \text{ and} \\
&I_i^j \cap \mathrm{Inf}(\rho) \neq \emptyset \text{ for every } j \in \{1, \ldots, \ell_i\}
\end{aligned}$$

Each $(F_i, \mathcal{I}_i) = \left(F_i, \{I_i^1, \ldots, I_i^{\ell_i}\}\right)$ is called a generalized Rabin pair (GRP)*, and the* GRP condition *is thus a disjunction of generalized Rabin pairs..*

W.l.o.g. we assume $k > 0$ and $\ell_i > 0$ for each $i \in \{1, \ldots, k\}$ (whenever $\ell_i = 0$ we could set $\mathcal{I}_i = \{Q\}$). Although the type of the condition allows for huge instances of the condition, the construction of [KE12] (producing this disjunctive normal form) guarantees efficiency not worse than that of the traditional determinization approach. For a formula of size n, it is guaranteed that $k \leq 2^n$ and $\ell_i \leq n$ for each $i \in \{1, \ldots, k\}$. Further, the size of the state space is at most $2^{\mathcal{O}(2^n)}$. Moreover, consider *"infinitary"* formulae, where each atomic proposition has both **F** and **G** as ancestors in the syntactic tree of the formula. Since the first component of the state space is always the same, the size of the state space is bounded by $2^{|Ap|}$ as the automaton only remembers the last letter read. We will make use of this fact later.

2.4 Degeneralization

As already discussed, one can blow up any automaton with generalized Rabin pairs and obtain a Rabin automaton. We need the following notation. For any $n \in \mathbb{N}$, let $[1..n]$ denote the set $\{1, \ldots, n\}$ equipped with the operation $\oplus$ of cyclic addition, i.e. $m \oplus 1 = m + 1$ for $m < n$ and $n \oplus 1 = 1$.

The DGRW defined above can now be degeneralized as follows. For each $i \in \{1, \ldots, k\}$, multiply the state space by $[1..\ell_i]$ to keep track for which I_i^j we are currently waiting for. Further, adjust the transition function so that we leave the jth copy once we visit I_i^j and immediately go to the next copy. Formally, for $\sigma \in \Sigma$ set $(q, w_1, \ldots, w_k) \xrightarrow{\sigma} (r, w_1', \ldots, w_k')$ if $q \xrightarrow{\sigma} r$ and $w_i' = w_i$ for all i with $q \notin I_i^{w_i}$ and $w_i' = w_i \oplus 1$ otherwise.

The resulting blow-up factor is then the following:

Definition 6 (Degeneralization index). *For a GRP condition* $\mathcal{GR} = \{(F_i, \mathcal{I}_i) \mid i \in [1..k]\}$*, we define the* degeneralization domain $B := \prod_{i=1}^{k} [1..|\mathcal{I}_i|]$ *and the* degeneralization index of $\mathcal{GR}$ *to be* $|B| = \prod_{i=1}^{k} |\mathcal{I}_i|$.

The state space of the resulting Rabin automaton is thus $|B|$-times bigger and the number of pairs stays the same. Indeed, for each $i \in \{1, \ldots, k\}$ we have a Rabin pair

$$\left(F_i \times B, I_i^{\ell_i} \times \{b \in B \mid b(i) = \ell_i\}\right)$$

Example 7. In Example 3 there is one pair and the degeneralization index is 2.

Example 8. For a conjunction of three fairness constraints $\varphi = (\mathbf{FG}a \vee \mathbf{GF}b) \wedge (\mathbf{FG}c \vee \mathbf{GF}d) \wedge (\mathbf{FG}e \vee \mathbf{GF}f)$, the Büchi components $\mathcal{I}_i$'s of the equivalent GRP condition correspond to $\mathbf{tt}, b, d, f, b \wedge d, b \wedge f, d \wedge f, b \wedge d \wedge f$. The degeneralization index is thus $|B| = 1 \cdot 1 \cdot 1 \cdot 1 \cdot 2 \cdot 2 \cdot 2 \cdot 3 = 24$. For four constraints, it is $1 \cdot 1^4 \cdot 2^6 \cdot 3^4 \cdot 4 = 20736$. One can easily see the index grows doubly exponentially.

3 Probabilistic Model Checking

In this section, we show how automata with generalized Rabin pairs can significantly speed up model checking of Markov decision processes (i.e., probabilistic model checking). For example, for the fairness constraints of the type mentioned in Example 8 the speed-up is by a factor that is doubly exponential. Although there are specialized algorithms for checking properties under strong fairness constraints (implemented in PRISM), our approach is general and speeds up for a wide class of constraints. The combinations (conjunctions, disjunctions) of properties not expressible by small Rabin automata (and/or Streett automata) are infeasible for the traditional approach, while we show that automata with generalized Rabin pairs often allow for efficient model checking. First, we present the theoretical model-checking algorithm for the new type of automata and the theoretical bounds for savings. Second, we illustrate the effectiveness of the approach experimentally.

3.1 Model Checking Using Generalized Rabin Pairs

We start with the definitions of Markov decision processes (MDPs), and present the model-checking algorithms. For a finite set V, let $\mathrm{Distr}(V)$ denote the set of probability distributions on V.

Definition 9 (MDP and MEC). *A* Markov decision process (MDP) $\mathcal{M} = (V, E, (V_0, V_P), \delta)$ *consists of a finite directed* MDP graph (V, E)*, a partition* (V_0, V_P) *of the* finite *set* V *of vertices into player-0 vertices* (V_0) *and probabilistic vertices* (V_P)*, and a probabilistic transition function* $\delta\colon V_P \to \mathrm{Distr}(V)$ *such that for all vertices* $u \in V_P$ *and* $v \in V$ *we have* $(u, v) \in E$ *iff* $\delta(u)(v) > 0$*.*

An end-component U *of an MDP is a set of its vertices such that (i) the subgraph induced by* U *is strongly connected and (ii) for each edge* $(u, v) \in E$*, if* $u \in U \cap V_P$*, then* $v \in U$ *(i.e., no probabilistic edge leaves* U*).*

A maximal end-component (MEC) *is an end-component that is maximal w.r.t. to the inclusion ordering.*

If U_1 and U_2 are two end-components and $U_1 \cap U_2 \neq \emptyset$, then $U_1 \cup U_2$ is also an end-component. Therefore, every MDP induces a unique set of its MECs, called *MEC decomposition*.

For precise definition of semantics of MDPs we refer to [Put94]. Note that MDPs are also defined in an equivalent way in literature with a set of actions such that every vertex and choice of action determines the probability distribution over the successor states; the choice of actions corresponds to the choice of edges at player-0 vertices of our definition.

The standard model-checking algorithm for MDPs proceeds in several steps. Given an MDP $\mathcal{M}$ and an LTL formula φ

1. compute a deterministic automaton $\mathcal{A}$ recognizing the language of φ,
2. compute the product $\overline{\mathcal{M}} = \mathcal{M} \times \mathcal{A}$,
3. solve the product MDP $\overline{\mathcal{M}}$.

The algorithm is generic for all types of deterministic ω-automata $\mathcal{A}$. The leading probabilistic model checker PRISM [KNP11] re-implements ltl2dstar [Kle] that transforms φ into a deterministic *Rabin* automaton. This approach employs Safra's determinization and thus despite many optimization often results in an unnecessarily big automaton.

There are two ways to fight the problem. Firstly, one can strive for smaller Rabin automata. Secondly, one can employ other types of ω-automata. As to the former, we have plugged our implementation Rabinizer [GKE12] of the approach [KE12] into PRISM, which already results in considerable improvement. For the latter, Example 4 shows that Muller automata can be smaller than Rabin automata. However, explicit representation of Muller acceptance conditions is typically huge. Hence the third step to solve the product MDP would be too expensive. Therefore, we propose to use automata with generalized Rabin pairs.

On the one hand, DGRW often have small state space after translation. Actually, it is the same as the state space of the intermediate Muller automaton of [KE12]. Compared to the corresponding naively degeneralized DRW it is $|B|$

times smaller (one can still perform some optimizations in the degeneralization process, see the experimental results).

On the other hand, as we show below the acceptance condition is still algorithmically efficient to handle. We now present the steps to solve the product MDP for a GRP acceptance condition, i.e. a disjunction of generalized Rabin pairs. Consider an MDP with k generalized Rabin pairs $(F_i, \{I_i^1, \ldots, I_i^{\ell_i}\})$, for $i = 1, 2, \ldots, k$. The steps of the computation are as follows:

1. For $i = 1, 2, \ldots, k$;
   (a) Remove the set of states F_i from the MDP.
   (b) Compute the MEC decomposition.
   (c) If a MEC C has a non-empty intersection with each I_i^j, for $j = 1, 2, \ldots, \ell_i$, then include C as a winning MEC.
   (d) let W_i be the union of winning MECs (for the ith pair).
2. Let W be the union of W_i, i.e. $W = \bigcup_{i=1}^{k} W_i$.
3. The solution (or optimal value of the product MDP) is the maximal probability to reach the set W.

Given an MDP with n vertices and m edges, let $\mathsf{MEC}(n, m)$ denote the complexity of computing the MEC decomposition; and $\mathsf{LP}(n, m)$ denotes the complexity to solve linear-programming solution with m constraints over n variables.

Theorem 10. *Given an MDP with n vertices and m edges with k generalized Rabin pairs $(F_i, \{I_i^1, \ldots, I_i^{\ell_i}\})$, for $i = 1, 2, \ldots, k$, the solution can be achieved in time $\mathcal{O}(k \cdot \mathsf{MEC}(n, m) + n \cdot \sum_{i=1}^{k} \ell_i) + \mathcal{O}(\mathsf{LP}(n, m))$.*

Remark 11. The best known complexity to solve MDPs with Rabin conditions of k pairs require time $\mathcal{O}(k \cdot \mathsf{MEC}(n, m)) + \mathcal{O}(\mathsf{LP}(n, m))$ time [dA97]. Thus degeneralization of generalized Rabin pairs to Rabin conditions and solving MDPs would require time $\mathcal{O}(k \cdot \mathsf{MEC}(|B| \cdot n, |B| \cdot m)) + \mathcal{O}(\mathsf{LP}(|B| \cdot n, |B| \cdot m))$ time. The current best known algorithms for maximal end-component decomposition require at least $\mathcal{O}(m \cdot n^{2/3})$ time [CH11], and the simplest algorithms that are typically implemented require $\mathcal{O}(n \cdot m)$ time. Thus our approach is more efficient at least by a factor of $B^{5/3}$ (given the current best known algorithms), and even if both maximal end-component decomposition and linear-programming can be solved in linear time, our approach leads to a speed-up by a factor of $|B|$, i.e. exponential in $\mathcal{O}(k)$ the number of non-trivially generalized Rabin pairs. In general if $\beta \geq 1$ is the sum of the exponents required to solve the MEC decomposition (resp. linear-programming), then our approach is better by a factor of $|B|^{\beta}$.

Example 12. A Rabin automaton for n constraints of Example 8 is of doubly exponential size, which is also the factor by which the product and thus the running time grows. However, as the formula is "infinitary" (see end of Section 2.3), the state space of the generalized automaton is 2^{Ap} and the product is of the very same size as the original system since the automaton only monitors the current labelling of the state.

3.2 Experimental Results

In this section, we compare the performance of

L the original PRISM with its implementation of ltl2dstar producing Rabin automata,

R PRISM with Rabinizer [GKE12] (our implementation of [KE12]) producing DRW via *optimized* degeneralization of DGRW, and

GR PRISM with Rabinizer producing DGRW and with the modified MEC checking step.

We have performed a case study on the Pnueli-Zuck randomized mutual exclusion protocol [PZ86] implemented as a PRISM benchmark. We consider the protocol with 3, 4, and 5 participants. The sizes of the respective models are $s_3 = 2\,368$, $s_4 = 27\,600$, and $s_5 = 308\,800$ states. We have checked these models against several formulae illustrating the effect of the degeneralization index on the speed up of our method; see Table 1.

In the first column, there are the formulae in the form of a PRISM query. We ask for a maximal/minimal value over all schedulers. Therefore, in the P_{max} case, we create an automaton for the formula, whereas in the case of P_{min} we create an automaton for its negation. The second column then states the number i of participants, thus inducing the respective size s_i of the model.

The next three columns depict the size of the product of the system and the automaton, for each of the **L**, **R**, **GR** variants. The size is given as the ratio of the actual size and the respective s_i. The number then describes also the "effective" size of the automaton when taking the product. The next three columns display the total running times for model checking in each variant.

The last three columns illustrate the efficiency of our approach. The first column $t_{\mathbf{R}}/t_{\mathbf{GR}}$ states the time speed-up of the DGRW approach when compared to the corresponding degeneralization. The second column states the degeneralization index $|B|$. The last column $t_{\mathbf{L}}/t_{\mathbf{GR}}$ then displays the overall speed-up of our approach to the original PRISM.

In the formulae, an atomic proposition $p_i = j$ denotes that the ith participant is in its state j. The processes start in state 0. In state 1 they want to enter the critical section. State 10 stands for being in the critical section. After leaving the critical section, the process re-enters state 0 again.

Formulae 1 to 3 illustrate the effect of $|B|$ on the ratio of sizes of the product in the **R** and **GR** cases, see $\frac{s_{\mathbf{R}}}{s_i}$, and ratio of the required times. The theoretical prediction is that $s_{\mathbf{R}}/s_{\mathbf{GR}} = |B|$. Nevertheless, due to optimizations done in the degeneralization process, the first is often slightly smaller than the second one, see columns $\frac{s_{\mathbf{R}}}{s_i}$ and B. (Note that $s_{\mathbf{GR}}/s_i$ is 1 for "infinitary" formulae.) For the same reason, $\frac{t_{\mathbf{R}}}{t_{\mathbf{GR}}}$ is often smaller than $|B|$. However, with the growing size of the systems it gets bigger hence the saving factor is larger for larger systems, as discussed in the previous section.

Formulae 4 to 7 illustrate the doubly exponential growth of $|B|$ and its impact on systems of different sizes. The DGRW approach (**GR** method) is often the only way to create the product at all.

Table 1. Experimental comparison of **L**, **R**, and **GR** methods. All measurements performed on Intel i7 with 8 GB RAM. The sign "−" denotes either crash, out-of-memory, time-out after 30 minutes, or a ratio where one operand is −.

Formula	#	$\frac{s_{\mathbf{L}}}{s_i}$	$\frac{s_{\mathbf{R}}}{s_i}$	$\frac{s_{\mathbf{GR}}}{s_i}$	$t_{\mathbf{L}}$	$t_{\mathbf{R}}$	$t_{\mathbf{GR}}$	$\frac{t_{\mathbf{R}}}{t_{\mathbf{GR}}}$	$\lvert B\rvert$	$\frac{t_{\mathbf{L}}}{t_{\mathbf{GR}}}$
P_{max} =?[**GF**p_1=10	3	4.1	2.6	1	1.2	0.4	0.2	2.2	3	6.8
∧ **GF**p_2=10	4	4.3	2.7	1	17.4	1.8	0.3	6.4	3	60.8
∧ **GF**p_3=10]	5	4.4	2.7	1	257.5	15.2	0.6	26.7	3	447.9
P_{max} =?[**GF**p_1=10 ∧ **GF**p_2=10	4	6	3.5	1	27.3	2.5	0.9	2.8	4	32.1
∧ **GF**p_3=10 ∧ **GF**p_4=10]	5	6.2	3.6	1	408.5	17.8	0.9	20.4	4	471.2
P_{min} =?[**GF**p_1=10 ∧ **GF**p_2=10	4	−	1	1	−	36.5	36.3	1	1	−
∧ **GF**p_3=10 ∧ **GF**p_4=10]	5	−	1	1	−	610.6	607.2	1	1	−
P_{max} =?[(**GF**p_1=0 ∨ **FG**p_2≠0)	3	79.7	1.9	1	225.5	4.1	2.2	1.8	2	101.8
∧(**GF**p_2=0 ∨ **FG**p_3≠0)]	4	−	1.9	1	−	61.7	29.2	2.1	2	−
	5	−	1.9	1	−	1007	479	2.1	2	−
P_{max} =?[(**GF**p_1=0 ∨ **FG**p_1≠0)	3	23.3	1.9	1	66.4	3.92	2.2	1.8	2	30.7
∧(**GF**p_2=0 ∨ **FG**p_2≠0)]	4	23.3	1.9	1	551.5	61	28.2	2.2	2	19.6
	5	−	1.9	1	−	1002.7	463	2.2	2	−
P_{max} =?[(**GF**p_1=0 ∨ **FG**p_2≠0)	3	−	16.3	1	−	122.1	7.1	17.2	24	−
∧(**GF**p_2=0 ∨ **FG**p_3≠0)	4	−	−	1	−	−	75.6	−	24	−
∧(**GF**p_3=0 ∨ **FG**p_1≠0)]	5	−	−	1	−	−	1219.5	−	24	−
P_{max} =?[(**GF**p_1=0 ∨ **FG**p_1≠0)	3	−	12	1	−	76.3	7.2	12	24	−
∧(**GF**p_2=0 ∨ **FG**p_2≠0)	4	−	12.1	1	−	1335.6	78.9	19.6	24	−
∧(**GF**p_3=0 ∨ **FG**p_3≠0)]	5	−	−	1	−	−	1267.6	−	24	−
P_{min} =?[(**GF**p_1≠10 ∨ **GF**p_1=0 ∨ **FG**p_1=1)	3	2.1	1	1	1.2	0.9	0.8	1	1	1.5
∧**GF**p_1≠0 ∧ **GF**p_1=1]	4	2.1	1	1	11.8	8.7	8.8	1	1	1.3
	5	2.1	1	1	186.3	147.5	146.2	1	1	1.3
P_{max} =?[(**G**p_1≠10 ∨ **G**p_2≠10 ∨ **G**p_3≠10)	3	−	32	5.9	−	405	80.1	5.1	8	−
∧ (**FG**p_1≠1 ∨ **GF**$p_2 = 1$ ∨ **GF**$p_3 = 1$)	4	−	−	6.4	−	−	703.5	−	8	−
∧ (**FG**p_2≠1 ∨ **GF**$p_1 = 1$ ∨ **GF**$p_3 = 1$)	5	−	−	−	−	−	−	−	8	−
P_{min} =?[(**FG**p_1≠0 ∨ **FG**p_2≠0 ∨ **GF**p_3=0)	3	55.9	4.7	1	289.7	12.6	3.4	3.7	12	84.3
∨ (**FG**p_1≠10 ∧ **GF**$p_2 = 10$ ∧ **GF**$p_3 = 10$)	4	−	4.6	1	−	194.5	33.2	5.9	12	−
	5	−	−	1	−	−	543	−	12	−

Formula 8 is a Streett condition showing the approach still performs competitively. Formulae 9 and 10 combine Rabin and Streett condition requiring both big Rabin automata and big Streett automata. Even in this case, the method scales well. Further, Formula 9 contains non-infinitary behaviour, e.g. **G**p_1≠10. Therefore, the DGRW is of size greater than 1, and thus also the product is bigger as can be seen in the $s_{\mathbf{GR}}/s_i$ column.

4 Synthesis

In this section, we show how generalized Rabin pairs can be used to speed up the computation of a winning strategy in an LTL(**F**,**G**) game and thus to speed up LTL(**F**,**G**) synthesis. A game is defined like an MDP, but with the stochastic vertices replaced by vertices of an adversarial player.

Definition 13. *A* game $\mathcal{M} = (V, E, (V_0, V_1))$ *consists of a finite directed* game graph (V, E) *and a partition* (V_0, V_1) *of the* finite *set* V *of vertices into player-0 vertices* (V_0) *and player-1 vertices* (V_1).

An *LTL game* is a game together with an LTL formula with vertices as atomic propositions. Similarly, a *Rabin game* and a *game with GRP condition (GRP game)* is a game with a set of Rabin pairs, or a set of generalized Rabin pairs, respectively.

A *strategy* is a function $V^* \to E$ assigning to each *history* an outgoing edge of its last vertex. A play conforming to the strategy f of Player 0 is any infinite sequence $v_0 v_1 \cdots$ satisfying $v_{i+1} = f(v_0 \cdots v_i)$ whenever $v_i \in V_0$, and just $(v_i, v_{i+1}) \in E$ otherwise. Player 0 has a *winning strategy*, if there is a strategy f such that all plays conforming to f of Player 0 satisfy the LTL formula, Rabin condition or GRP condition, depending on the type of the game. For further details, we refer to e.g. [PP06].

One way to solve an LTL game is to make a product of the game arena with the DRW corresponding to the LTL formula, yielding a Rabin game. The current fastest solution of Rabin games works in time $\mathcal{O}(mn^{k+1}kk!)$ [PP06], where $n = |V|, m = |E|$ and k is the number of pairs. Since n is doubly exponential and k singly exponential in the size of the formula, this leads to a doubly exponential algorithm. And indeed, the problem of LTL synthesis is 2-EXPTIME-complete [PR89].

Similarly as for model checking of probabilistic systems, we investigate what happens (1) if we replace the translation to Rabin automata by our new translation and (2) if we employ DGRW instead. The latter leads to the problem of GRP games. In order to solve them, we extend the methods to solve Rabin and Streett games of [PP06].

We show that solving a GRP game is faster than first degeneralizing them and then solving the resulting Rabin game. The induced speed-up factor is $|B|^k$. In the following two subsections we show how to solve GRP games and analyze the complexity. The subsequent section reports on experimental results.

4.1 Generalized Rabin Ranking

We shall compute a ranking of each vertex, which intuitively states how far from winning we are. The existence of winning strategy is then equivalent to the existence of a ranking where Player 0 can always choose a successor of the current vertex with smaller ranking, i.e. closer to fulfilling the goal.

Let $(V, E, (V_0, V_1))$ be a game, $\{(F_1, \mathcal{I}_1), \ldots, (F_k, \mathcal{I}_k)\}$ a GRP condition with the corresponding degeneralization domain B. Further, let $n := |V|$ and denote the set of permutations over a set S by $S!$.

Definition 14. *A* ranking *is a function* $r : V \times B \to R$ *where* R *is the* ranking domain $\{1, \ldots, k\}! \times \{0, \ldots, n\}^{k+1} \cup \{\infty\}$.

The ranking $r(v, wf)$ gives information important in the situation when we are in vertex v and are waiting for a visit of $\mathcal{I}_i^{wf(i)}$ for each i given by $wf \in B$. As time passes the ranking should decrease. To capture this, we define the following functions.

Definition 15. *For a ranking r and given $v \in V$ and $wf \in B$, we define* $\mathrm{next}_v : B \to B$

$$\mathrm{next}_v(wf)(i) = \begin{cases} wf(i) & \text{if } v \notin \mathcal{I}_i^{wf(i)} \\ wf(i) \oplus 1 & \text{if } v \in \mathcal{I}_i^{wf(i)} \end{cases}$$

and $\mathrm{next} : V \times B \to R$

$$\mathrm{next}(v, wf) = \begin{cases} \min_{(v,w)\in E} r(w, \mathrm{next}_v(wf)) & \text{if } v \in V_0 \\ \max_{(v,w)\in E} r(w, \mathrm{next}_v(wf)) & \text{if } v \in V_1 \end{cases}$$

where the order on $(\pi_1 \cdots \pi_k, w_0 w_1 \cdots w_k) \in R$ is given by the lexicographic order $>_{lg}$ on $w_0\pi_1 w_1 \pi_2 w_2 \cdots \pi_k w_k$ and ∞ being the greatest element.

Intuitively, the ranking $r(v, wf) = (\pi_1 \cdots \pi_k, w_0 w_1 \cdots w_k)$ is intended to bear the following information. The permutation π states the importance of the pairs. The pair $(F_{\pi_1}, \mathcal{I}_{\pi_1})$ is the most important, hence we are not allowed to visit F_{π_1} and we desire to either visit $\mathcal{I}_{\pi_1}$, or not visit F_{π_2} and visit $\mathcal{I}_{\pi_2}$ and so on. If some important F_i is visited it becomes less important. The importance can be freely changed only finitely many (i_0) times. Otherwise, only less important pairs can be permuted if a more important pair makes good progress. Further, w_i measures the worst possible number of steps until visiting $\mathcal{I}_{\pi_i}$. This intended meaning is formalized in the following notion of good rankings.

Definition 16. *A ranking r is* good *if for every $v \in V, wf \in B$ with $r(v, wf) \neq \infty$ we have $r(v, wf) >_{v,wf} \mathrm{next}(v, wf)$.*

We define $(\pi_1 \cdots \pi_k, w_0 w_1 \cdots w_k) >_{v,wf} (\pi'_1 \cdots \pi'_k, w'_0 w'_1 \cdots w'_k)$ if either $w_0 > w'_0$, or $w_0 = w'_0$ with $>^1_{v,wf}$ hold. Recursively, $>^\ell_{v,wf}$ holds if one of the following holds:

- $\pi_\ell > \pi'_\ell$
- $\pi_\ell = \pi'_\ell$, $v \not\models F_{\pi_\ell}$ *and* $w_\ell > w'_\ell$
- $\pi_\ell = \pi'_\ell$, $v \not\models F_{\pi_\ell}$ *and* $v \models \mathcal{I}^{wf(\pi_\ell)}_{\pi_\ell}$
- $\pi_\ell = \pi'_\ell$, $v \not\models F_{\pi_\ell}$ *and* $w_\ell = w'_\ell$ *and* $>^{\ell+1}_{v,wf}$ *holds (where* $>^{k+1}_{v,wf}$ *never holds)*

Moreover, if one of the first three cases holds, we say that $\succ^\ell_{v,wf}$ holds.

Intuitively, $>$ means the second element is closer to the next milestone and $\succ^\ell$, moreover, that it is so because of the first ℓ pairs in the permutation.

Similarly to [PP06], we obtain the following correctness of the construction. Note that for $|B| = 1$, the definitions of the ranking here and the *Rabin ranking* of [PP06] coincide. Further, the extension with $|B| > 1$ bears some similarities with the Streett ranking of [PP06].

Theorem 17. *For every vertex v, Player 0 has a winning strategy from v if and only if there is a good ranking r and $wf \in B$ with $r(v, wf) \neq \infty$.*

4.2 A Fixpoint Algorithm

In this section, we show how to compute the smallest good ranking and thus solve the GRP game. Consider a lattice of rankings ordered component-wise, i.e. $r_1 >_c r_2$ if for every $v \in V$ and $wf \in B$, we have $r_1(v, wf) >_{lg} r_2(v, wf)$. This induces a complete lattice. The minimal good ranking is then a least fixpoint of the operator Lift on rankings given by:

$$\text{Lift}(r)(v, wf) = \max\big\{r(v, wf), \min\{x \mid x >_{v,wf} \text{next}(v, wf)\}\big\}$$

where the optima are considered w.r.t. $>_{lg}$. Intuitively, if Player 0 cannot choose a successor smaller than the current vertex (or all successors of a Player 1 vertex are greater), the ranking of the current vertex must rise so that it is greater.

Theorem 18. *The smallest good ranking can be computed in time* $\mathcal{O}(mn^{k+1}kk! \cdot |B|)$ *and space* $(nk \cdot |B|)$.

Proof. The lifting operator can be implemented similarly as in [PP06]. With every change, the affected predecessors to be updated are put in a worklist, thus working in time $\mathcal{O}(k \cdot \text{out-deg}(v))$. Since every element can be lifted at most $|R|$-times, the total time is $\mathcal{O}(\sum_{v \in V} \sum_{wf \in B} k \cdot \text{out-deg}(v) \cdot |R|) = |B|km \cdot n^{k+1}k!$. The space required to store the current ranking is $\mathcal{O}(\sum_{v \in V} \sum_{wf \in B} k) = n \cdot |B| \cdot k$. □

We now compare our solution to the one that would solve the degeneralized Rabin game. The number of vertices of the degeneralized Rabin game is $|B|$ times greater. Hence the time needed is multiplied by a factor $|B|^{k+2}$, instead of $|B|$ in the case of a GRP game. Therefore, our approach speeds up by a factor of $|B|^{k+1}$, while the space requirements are the same in both cases, namely $\mathcal{O}(nk \cdot |B|)$.

Example 19. A conjunction of two fairness constraints of example 8 corresponds to $|B| = 2$ and $k = 4$, hence we save by a factor of $2^4 = 16$. A conjunction of three fairness constraints corresponds to $|B| = 24$ and $k = 8$, hence we accelerate $24^8 \approx 10^{11}$ times.

Further, let us note that the computation can be implemented recursively as in [PP06]. The winning set is $\mu Z.\, \mathfrak{GR}(\mathcal{GR}, \mathtt{tt}, \heartsuit Z)$ where $\mathfrak{GR}(\emptyset, \varphi, W) = W$,

$$\mathfrak{GR}(\mathcal{GR}, \varphi, W) = \bigvee_{i \in [1..k]} \nu Y.\, \bigwedge_{j \in [1..|\mathcal{I}_i|]} \mu X.\, \mathfrak{GR}\Big(\mathcal{GR} \setminus \{(F_i, \mathcal{I}_i)\}, \varphi \wedge \neg F_i, W \vee (\varphi \wedge \neg F_i \wedge I_i^j \wedge \heartsuit Y) \vee (\varphi \wedge \neg F \wedge \heartsuit X)\Big)$$

$\heartsuit\varphi = \{u \in V_0 \mid \exists (u, v) \in E : v \models \varphi\} \cup \{u \in V_1 \mid \forall (u, v) \in E : v \models \varphi\}$ and μ and ν denote the least and greatest fixpoints, respectively. The formula then provides a succinct description of a symbolic algorithm.

4.3 Experimental Evaluation

Reusing the notation of Section 3.2, we compare the performance of the methods for solving LTL games. We build and solve a Rabin game using

L ltl2dstar producing DRW (from LTL formulae),
R Rabinizer producing DRW, and
GR Rabinizer producing DGRW.

We illustrate the methods on three different games and three LTL formulae; see Table 2. The games contain 3 resp. 6 resp. 9 vertices. Similarly to Section 3.2, s_i denotes the number of vertices in the ith arena, $s_{\mathbf{L}}, s_{\mathbf{R}}, s_{\mathbf{GR}}$ the number of vertices in the resulting games for the three methods, and $t_{\mathbf{L}}, t_{\mathbf{R}}, t_{\mathbf{GR}}$ the respective running times.

Formula 1 allows for a winning strategy and the smallest ranking is relatively small, hence computed quite fast. Formula 2, on the other hand, only allows for larger rankings. Hence the computation takes longer, but also because in **L** and **R** cases the automata are larger than for formula 1. While for **L** and **R**, the product is usually too big, there is a chance to find small rankings in **GR** fast. While for e.g. $\mathbf{FG}(a \vee \neg b \vee c)$, the automata and games would be the same for all three methods and the solution would only take less than a second, the more complex formulae 1 and 2 show clearly the speed up.

Table 2. Experimental comparison of **L**, **R**, and **GR** methods for solving LTL games. Again the sign "−" denotes either crash, out-of-memory, time-out after 30 minutes, or a ratio where one operand is −.

Formula	s_i	$\frac{s_{\mathbf{L}}}{s_i}$	$\frac{s_{\mathbf{R}}}{s_i}$	$\frac{s_{\mathbf{GR}}}{s_i}$	$t_{\mathbf{L}}$	$t_{\mathbf{R}}$	$t_{\mathbf{GR}}$	$\frac{t_{\mathbf{R}}}{t_{\mathbf{GR}}}$	$\lvert B\rvert$	$\frac{t_{\mathbf{L}}}{t_{\mathbf{GR}}}$
$(\mathbf{GF}a \wedge \mathbf{GF}b \wedge \mathbf{GF}c)$	3	22	7.3	4	63.2	1.6	1.1	1.4	9	48.2
$\vee(\mathbf{GF}\neg a \wedge \mathbf{GF}\neg b \wedge \mathbf{GF}\neg c)$	6	21.3	7.3	3.7	878.6	14.1	7.3	2	9	130.3
	9	20.6	7	3.6	−	54.8	31.3	1.8	9	−
$(\mathbf{GF}a \vee \mathbf{FG}b) \wedge (\mathbf{GF}c \vee \mathbf{GF}\neg a)$	3	21	10	4	−	117.5	12	9.8	6	−
$\wedge(\mathbf{GF}c \vee \mathbf{GF}\neg b)$	6	16.2	9.2	3.7	−	−	196.7	−	6	−
	9	17.6	9.2	3.6	−	−	1017.8	−	6	−

5 Conclusions

In this work we considered the translation of the LTL(**F**,**G**) fragment to deterministic ω-automata that is necessary for probabilistic model checking as well as synthesis. The direct translation to deterministic Muller automata gives a compact automata but the explicit representation of the Muller condition is huge and not algorithmically amenable. In contrast to the traditional approach of translation to deterministic Rabin automata that admits efficient algorithms but incurs a blow-up in translation, we consider deterministic automata with generalized Rabin pairs (DGRW). The translation to DGRW produces the same compact automata as for Muller conditions. We presented efficient algorithms for probabilistic model checking and game solving with DGRW conditions which shows

that the blow-up of translation to Rabin automata is unnecessary. Our results establish that DGRW conditions provide the convenient formalism that allows both for compact automata as well as efficient algorithms. We have implemented our approach in PRISM, and experimental results show a huge improvement over the existing methods. Two interesting directions of future works are (1) extend our approach to LTL with the **U**(until) and the **X**(next) operators; and (2) consider symbolic computation and Long's acceleration of fixpoint computation (on the recursive algorithm), instead of the ranking function based algorithm for games, and compare the efficiency of both the approaches.

References

[AT04] Alur, R., La Torre, S.: Deterministic generators and games for LTL fragments. ACM Trans. Comput. Log. 5(1), 1–25 (2004)

[BK08] Baier, C., Katoen, J.-P.: Principles of model checking. MIT Press (2008)

[CH11] Chatterjee, K., Henzinger, M.: Faster and dynamic algorithms for maximal end-component decomposition and related graph problems in probabilistic verification. In: SODA, pp. 1318–1336 (2011)

[Chu62] Church, A.: Logic, arithmetic, and automata. In: Proceedings of the International Congress of Mathematicians, pp. 23–35. Institut Mittag-Leffler (1962)

[CY95] Courcoubetis, C., Yannakakis, M.: The complexity of probabilistic verification. J. ACM 42(4), 857–907 (1995)

[dA97] de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University (1997)

[Ehl11] Ehlers, R.: Generalized rabin(1) synthesis with applications to robust system synthesis. In: NASA Formal Methods, pp. 101–115 (2011)

[GKE12] Gaiser, A., Křetínský, J., Esparza, J.: Rabinizer: Small deterministic automata for lTL(**F**,**G**). In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 72–76. Springer, Heidelberg (2012)

[JB06] Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society (2006)

[JGWB07] Jobstmann, B., Galler, S., Weiglhofer, M., Bloem, R.: Anzu: A tool for property synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 258–262. Springer, Heidelberg (2007)

[KB06] Klein, J., Baier, C.: Experiments with deterministic ω-automata for formulas of linear temporal logic. Theor. Comput. Sci. 363(2), 182–195 (2006)

[KB07] Klein, J., Baier, C.: On-the-fly stuttering in the construction of deterministic ω-automata. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 51–61. Springer, Heidelberg (2007)

[KE12] Křetínský, J., Esparza, J.: Deterministic automata for the (F,G)-fragment of LTL. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 7–22. Springer, Heidelberg (2012)

[Kle] Klein, J.: ltl2dstar - LTL to deterministic Streett and Rabin automata, `http://www.ltl2dstar.de/`

[KNP11] Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)

[Kup12] Kupferman, O.: Recent challenges and ideas in temporal synthesis. In: Bieliková, M., Friedrich, G., Gottlob, G., Katzenbeisser, S., Turán, G. (eds.) SOFSEM 2012. LNCS, vol. 7147, pp. 88–98. Springer, Heidelberg (2012)

[KV05] Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–542. IEEE Computer Society (2005)

[MS08] Morgenstern, A., Schneider, K.: From LTL to symbolically represented deterministic automata. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 279–293. Springer, Heidelberg (2008)

[Pit06] Piterman, N.: From nondeterministic Buchi and Streett automata to deterministic parity automata. In: LICS, pp. 255–264 (2006)

[Pnu77] Pnueli, A.: The temporal logic of programs. In: FOCS, pp. 46–57. IEEE (1977)

[PP06] Piterman, N., Pnueli, A.: Faster solutions of rabin and streett games. In: LICS, pp. 275–284 (2006)

[PPS06] Piterman, N., Pnueli, A., Sa'ar, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2006)

[PR89] Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ronchi Della Rocca, S., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)

[Put94] Puterman, M.L.: Markov Decision Processes (1994)

[PZ86] Pnueli, A., Zuck, L.: Verification of multiprocess probabilistic protocols. Distributed Computing 1(1), 53–72 (1986)

[Saf88] Safra, S.: On the complexity of ω-automata. In: FOCS, pp. 319–327. IEEE Computer Society (1988)

[Var85] Vardi, M.Y.: Automatic verification of probabilistic concurrent finite-state programs. In: FOCS, pp. 327–338 (1985)

[VW86] Vardi, M.Y., Wolper, P.: Automata-theoretic techniques for modal logics of programs. J. Comput. Syst. Sci. 32(2), 183–221 (1986)

Importance Splitting for Statistical Model Checking Rare Properties

Cyrille Jegourel, Axel Legay, and Sean Sedwards

{cyrille.jegourel,axel.legay,sean.sedwards}@inria.fr

Abstract Statistical model checking avoids the intractable growth of states associated with probabilistic model checking by estimating the probability of a property from simulations. Rare properties are often important, but pose a challenge for simulation-based approaches: the relative error of the estimate is unbounded. A key objective for statistical model checking rare events is thus to reduce the variance of the estimator. *Importance splitting* achieves this by estimating a sequence of conditional probabilities, whose product is the required result. To apply this idea to model checking it is necessary to define a *score function* based on logical properties, and a set of *levels* that delimit the conditional probabilities.

In this paper we motivate the use of importance splitting for statistical model checking and describe the necessary and desirable properties of score functions and levels. We illustrate how a score function may be derived from a property and give two importance splitting algorithms: one that uses fixed levels and one that discovers optimal levels adaptively.

1 Introduction

Model checking offers the possibility to verify the correctness of complex systems in an automatic way [6]. This concept has now been extended to probabilistic systems, where some or all non-determinism is resolved to probabilities or stochastic rates [1]. This extension is of fundamental importance, since in many practical applications it is necessary to quantify the probability of a property (e.g., system failure) or the expectation of an amount (e.g., the yield of a process).

To give results with certainty, model checking algorithms effectively perform an exhaustive traversal of the state space of the system. In most real applications, however, the state space is intractable, scaling exponentially with the number of interacting components. Abstraction and symmetry reduction may make certain classes of systems tractable, but are not generally applicable. This limitation has prompted the development of *statistical* model checking, that employs an executable model of the system to estimate the probability of a property from a number of independent simulations.

Statistical model checking is a Monte Carlo method [17] that takes advantage of robust statistical techniques to bound the error of the estimated result (e.g., [5,23]). To quantify a property it is necessary to observe the property and increasing the number of observations generally increases the confidence of the estimate. Rare properties thus pose a problem to statistical model checking, since

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 576–591, 2013.

they are difficult to observe and often highly relevant to system performance (e.g., system failure is usually required to be rare). Fortunately, many Monte Carlo methods for rare events were devised in the early days of computing. In particular, *importance sampling* [13,15] and *importance splitting* [14,15,20] may be successfully applied to statistical model checking.

Importance sampling and importance splitting have been widely applied to specific simulation problems in science and engineering. Importance sampling works by estimating a result using biased simulations and compensating for the bias. Importance splitting works by reformulating the rare probability as a product of less rare probabilities conditioned on levels that must be achieved.

Earlier work [22,10] extended the original applications of importance splitting to more general problems of computational systems. Recent work has explicitly considered the use of importance sampling in the context of statistical model checking [18,11,2,12]. In what follows, we describe some of the limitations of importance sampling and motivate the use of importance splitting applied to statistical model checking, linking the concept of levels and score functions to temporal logic.

The remainder of the paper is organised as follows. Section 2 defines the basic notions and notation required in the sequel. Section 3 discusses statistical model checking rare events and introduces importance sampling and splitting. Section 4 defines the important properties of score functions (required to define levels) and describes how such functions may be derived from logical properties. Section 5 gives two importance splitting algorithms, while Section 6 illustrates their use on several examples, using different score functions.

2 Preliminaries

We consider stochastic discrete-event systems. This class includes any stochastic process that can be thought of as occupying a single state for a duration of time before an instantaneous transition to a new state. In particular, we consider systems described by discrete and continuous time Markov chains. Sample execution paths can be generated through discrete-event simulation (e.g., [9]). Execution paths are sequences of the form $\omega = s_0 \xrightarrow{t_0} s_1 \xrightarrow{t_1} s_2 \xrightarrow{t_2} ...$, where each $s_i \in S$ is a state of the model and $t_i \in \mathbb{R} > 0$ is the time spent in the state s_i (the delay time) before moving to the state s_{i+1}. In the case of discrete time, $t_i \equiv 1, \forall i$. When we are not interested by the times of jump epochs, we denote a path $\omega = s_0 s_1 ....$ The length of path ω includes the initial state and is denoted $|\omega|$. A prefix of ω is a sequence $\omega_{\leq k} = s_0 s_1 ... s_k$ with $k < |\omega| \in \mathbb{N}$. We denote by $\omega_{\geq k}$ the suffix of ω starting at s_k.

2.1 Temporal Logic

A simulation trace results from an execution of the system and is a finite sequence of visited states labelled with either discrete time step numbers or real times that are accumulated random delays. The discrete or continuous time

Markov chains that describe the system may be infinite. The process of statistical model checking estimates the probability that a system satisfies a property from the number of simulation traces within a sample, which individually satisfy the property. In this paper we consider properties specified with time bounded temporal logic, having the following abstract syntax:

$$\varphi = \alpha \mid \varphi \vee \varphi \mid \varphi \wedge \varphi \mid \neg\varphi \mid \mathbf{X}\varphi \mid \mathbf{F}^t\varphi \mid \mathbf{G}^t\varphi \mid \varphi\mathbf{U}^t\varphi \tag{1}$$

α is an *atomic proposition* that may be true (denoted $\top$) or false in any state $s \in S$. $\vee$, $\wedge$ and $\neg$ are the standard Boolean connectives. $\mathbf{F}^t$, $\mathbf{G}^t$ and $\mathbf{U}^t$ are temporal operators that apply to time interval $[0, t]$, where $t \in \mathbb{R}$ may denote steps or real time and the interval is relative to the interval of any enclosing operator. To simplify the following notation, it is assumed that if a property requires the next state to be satisfied and no next state exists, the property is not satisfied. Thus, given an arbitrary suffix $\omega_{\geq k}$ and a property φ with syntax (1), the semantics of $\omega_{\geq k} \models \varphi$ is defined:

- $\omega_{\geq k} \models \alpha \iff \alpha$ is true in state s_k
- $\omega_{\geq k} \models \varphi_1 \vee \varphi_2 \iff \omega_{\geq k} \models \varphi_1 \vee \omega_{\geq k} \models \varphi_2$
- $\omega_{\geq k} \models \varphi_1 \wedge \varphi_2 \iff \omega_{\geq k} \models \varphi_1 \wedge \omega_{\geq k} \models \varphi_2$
- $\omega_{\geq k} \models \neg\varphi \iff \omega_{\geq k} \not\models \varphi$
- $\omega_{\geq k} \models \mathbf{X}\varphi \iff \omega_{\geq k+1} \models \varphi$
- $\omega_{\geq k} \models \mathbf{F}^t\varphi \iff \exists i \geq k \in \mathbb{N} : \sum_{l\in\{k,\dots,i\}} t_l \leq t \wedge \omega_{\geq i} \models \varphi$
- $\omega_{\geq k} \models \mathbf{G}^t\varphi \iff \exists i \geq k \in \mathbb{N} : \sum_{l\in\{k,\dots,i\}} t_l \leq t \wedge \sum_{l\in\{k,\dots,i+1\}} t_l > t \wedge \forall l \in \{k,\dots,i\} : \omega_{\geq l} \models \varphi$
- $\omega_{\geq k} \models \varphi_1\mathbf{U}^t\varphi_2 \iff \exists i \geq k \in \mathbb{N} : \sum_{l\in\{k,\dots,i\}} t_l \leq t \wedge \omega_{\geq i} \models \varphi_2 \wedge (i = k \vee \forall l \in \{k,\dots,i-1\} : \omega_{\geq l} \models \varphi_1)$

Informally: $\mathbf{X}\varphi$ means that φ will be true in the next state; $\mathbf{F}^t\varphi$ means that φ will be true at least once in the interval $[0, t]$; $\mathbf{G}^t\varphi$ means that φ will always be true in the interval $[0, t]$; $\psi\mathbf{U}^t\varphi$ means that in the interval $[0, t]$, φ will eventually be true and ψ will be true until it is. $\mathbf{F}^t$, $\mathbf{G}^t$ and $\mathbf{U}^t$ are related in the following way: $\mathbf{G}^t = \neg(\mathbf{F}^t\neg\varphi)$, $\mathbf{F}^t\varphi = \top\mathbf{U}^t\varphi$, hence $\mathbf{G}^t\varphi = \neg(\top\mathbf{U}^t\neg\varphi)$.

3 Statistical Model Checking Rare Events

We consider a stochastic system $\mathcal{S}$ and a temporal logic property φ that may be true or false with respect to an execution trace. Our objective is to calculate the probability γ that an arbitrary execution trace ω satisfies φ, denoted $\gamma = \mathrm{P}(\omega \models \varphi)$. To decide the truth of a particular trace ω', we define a model checking function $z(\omega) \in \{0, 1\}$ that takes the value 1 if $\omega' \models \varphi$ and 0 if $\omega' \not\models \varphi$.

Let Ω be the set of paths induced by $\mathcal{S}$, with $\omega \in \Omega$ and f a probability measure over Ω. Then

$$\gamma = \int_\Omega z(\omega)\,\mathrm{d}f \qquad (2) \qquad \text{and} \qquad \gamma \approx \frac{1}{N}\sum_{i=1}^{N} z(\omega_i)$$

N denotes the number of simulations and ω_i is sampled according to f. Note that $z(\omega_i)$ is effectively the realisation of a Bernoulli random variable with parameter γ. Hence $\mathrm{Var}(\gamma) = \gamma(1-\gamma)$ and for $\gamma \to 0$, $\mathrm{Var}(\gamma) \approx \gamma$.

When a property is not rare there are useful bounding formulae (e.g., the Chernoff bound [5]) that relate *absolute* error, confidence and the required number of simulations to achieve them. As the property becomes rarer, however, absolute error ceases to be useful and it is necessary to consider *relative error*, defined as the standard deviation of the estimate divided by its expectation. For a Bernoulli random variable the relative error is given by $\sqrt{\gamma(1-\gamma)}/\gamma$, that is unbounded as $\gamma \to 0$. In standard Monte Carlo simulation, γ is the expected fraction of executions in which the rare event will occur. If the number of simulation runs is significantly less than $1/\gamma$, as is necessary when γ is very small, no occurrences of the rare property will likely be observed. A number of simulations closer to $100/\gamma$ is desirable to obtain a reasonable estimate. Hence, the following techniques have been developed to reduce the number of simulations required or, equivalently, to reduce the variance of the rare event and so achieve greater confidence for a given number of simulations.

3.1 Importance Sampling

Importance sampling works by biasing the system dynamics in favour of a property of interest, simulating under the new dynamics, then unbiasing the result to give a true estimate. Referring to (2), let f' be another probability measure over Ω, absolutely continuous with respect to zf, then (2) can be rewritten

$$\gamma = \int_\Omega z(\omega) \frac{\mathrm{d}f(\omega)}{\mathrm{d}f'(\omega)} \, \mathrm{d}f' = \int_\Omega L(\omega) z(\omega) \, \mathrm{d}f'$$

where $L = \mathrm{d}f/\mathrm{d}f'$ is the *likelihood ratio*. We can thus estimate γ by simulating under f' and compensating by L: $\gamma \approx \frac{1}{N} \sum_{i=1}^{N} L(\omega_i) z(\omega_i)$. $L(\omega_i)$ may be calculated with little overhead during individual simulation runs.

In general, the importance sampling measure f' is chosen to produce the rare property more frequently, but this is not the only criterion. The optimal importance sampling measure, denoted f^* and defined as f conditioned on the rare event, is exactly the distribution of the rare event: $f^* = zf/\gamma$. The challenge of importance sampling is to find a good *change of measure*, i.e., a measure f' that is close to f^*. An apparently good change of measure may produce the rare property more frequently (thus reducing the variance with respect to the *estimated* value) but increase the variance with respect to the *true* value. In [12] we describe an efficient algorithm to find a change of measure that avoids this phenomenon.

It remains an open problem with importance sampling to quantify the performance of apparently 'good' distributions. A further challenge arises from properties and systems that require long simulations. In general, as the length of a path increases, its probability diminishes exponentially, leading to very subtle differences between f and f' and consequent problems of numerical precision.

3.2 Importance Splitting

The earliest application of importance splitting is perhaps that of [14], where it was used to calculate the probability that neutrons would pass through certain shielding materials. This physical example provides a convenient analogy for the more general case. The system comprises a source of neutrons aimed at one side of a shield of thickness T. It is assumed that neutrons are absorbed by random interactions with the atoms of the shield, but with some small probability γ it is possible for a neutron to pass through the shield. The distance travelled in the shield can then be used to define a set of increasing levels $l_0 = 0 < l_1 < l_2 < \cdots < l_n = T$ that may be reached by the paths of neutrons, with the property that reaching a given level implies having reached all the lower levels. Though the overall probability of passing through the shield is small, the probability of passing from one level to another can be made arbitrarily close to 1 by reducing the distance between the levels.

These concepts can be generalised to simulation models of arbitrary systems, where a path is a simulation trace. By denoting the abstract level of a path as l, the probability of reaching level l_i can be expressed as $\mathrm{P}(l > l_i) = \mathrm{P}(l > l_i \mid l > l_{i-1})\mathrm{P}(l > l_{i-1})$. Defining $\gamma = \mathrm{P}(l > l_n)$ and observing $\mathrm{P}(l > l_0) = 1$, it is possible to write

$$\gamma = \prod_{i=1}^{n} \mathrm{P}(l > l_i \mid l > l_{i-1}) \tag{3}$$

Each term of the product (3) is necessarily greater than or equal to γ. The technique of importance splitting thus uses (3) to decompose the simulation of a rare event into a series of simulations of conditional events that are less rare. There have been many different implementations of this idea, but a generalised procedure is as follows.

Assuming a set of increasing levels is defined as above, a number of simulations are generated, starting from a distribution of initial states that correspond to reaching the current level. The procedure starts by estimating $\mathrm{P}(l \geq l_1 | l \geq l_0)$, where the distribution of initial states for l_0 is usually given (often a single state). Simulations are stopped as soon as they reach the next level; the final states becoming the empirical distribution of initial states for the next level. Simulations that do not reach the next level (or reach some other stopping criterion) are discarded. In general, $\mathrm{P}(l \geq l_i | l \geq l_{i-1})$ is estimated by the number of simulation traces that reach l_i, divided by the total number of traces started from l_{i-1}. Simulations that reached the next level are continued from where they stopped. To avoid a progressive reduction of the number of simulations, the generated distribution of initial states is sampled to provide additional initial states for new simulations, thus replacing those that were discarded.

In physical and chemical systems, distances and quantities may provide a natural notion of level that can be finely divided. In the context of model-checking arbitrary systems, variables may be Boolean and temporal properties may not contain an obvious notion of level. To apply importance splitting to statistical

model checking it is necessary to define a set of levels based on a sequence of temporal properties, φ_i, that have the logical characteristic

$$\varphi = \varphi_n \Rightarrow \varphi_{n-1} \Rightarrow \cdots \Rightarrow \varphi_0$$

Each φ_i is a strict restriction of the property φ_{i-1}, formed by the conjunction of φ_i with property ψ_i, such that $\varphi_i = \varphi_{i-1} \wedge \psi_i$, with $\varphi_0 \equiv \top$. Hence, φ_i can be written $\varphi_i = \bigwedge_{j=1}^{i} \psi_j$. This induces a strictly nested sequence of sets of paths $\Omega_i \subseteq \Omega$:

$$\Omega_n \subset \Omega_{n-1} \subset \cdots \subset \Omega_0$$

where $\Omega_i = \{\omega \in \Omega : \omega \models \varphi_i\}$, $\Omega_0 \equiv \Omega$ and $\forall \omega \in \Omega, \omega \models \varphi_0$. Thus, for arbitrary $\omega \in \Omega$,

$$\gamma = \prod_{i=1}^{n} \mathrm{P}(\omega \models \varphi_i \mid \omega \models \varphi_{i-1}),$$

that is analogous to (3).

A statistical model checker implementing bounded temporal logic will generally assign variables to track the status of the time bounds of temporal operators. Importance splitting requires these variables to be included as part of the state that is stored when a trace reaches a given level.

The choice of levels is crucial to the effectiveness of importance splitting. To minimise the relative variance of the final estimate it is desirable to choose levels that make $\mathrm{P}(\omega \models \varphi_i \mid \omega \models \varphi_{i-1})$ the same for all i (see, e.g., [7]). A simple decomposition of a property may give levels with widely divergent conditional probabilities, hence Section 4 introduces the concept of a *score function* and techniques that may be used to increase the possible resolution of levels. Given sufficient resolution, a further challenge is to define the levels. In practice, these are often guessed or found by trial and error, but Section 5.2 gives an algorithm that finds optimal levels adaptively.

4 Score Functions

Score functions generalise the concept of levels described in Section 3.2.

Definition 1. *Let $J_0 \supset J_1 \supset \ldots \supset J_n$ be a set of nested intervals of $\mathbb{R}$ and let $\varphi_0 \Leftarrow \varphi_1 \Leftarrow \cdots \Leftarrow \varphi_n = \varphi$ be a set of nested properties. The mapping $\Phi : \Omega \to \mathbb{R}$ is a* level-based *score function of property φ if and only if $\forall k : \omega \models \varphi_k \iff \Phi(\omega) \in J_k$ and $\forall i, j \in \{0, \ldots, |\omega|\} : i < j \implies \Phi(\omega_{\leq i}) \leq \Phi(\omega_{\leq j})$*

In general, the aim of a score function is to discriminate good paths from bad with respect to a property. In the case of a level-based score function, paths that have a higher score are clearly better because they satisfy more of the overall property. Given a nested sequence of properties $\varphi_0 = \top \Leftarrow \varphi_1 \Leftarrow \cdots \Leftarrow \varphi_n = \varphi$, a simple score function may be defined

$$\Phi(\omega) = \sum_{k=1}^{n} \mathbb{1}(\omega \models \varphi_k) \tag{4}$$

$\mathbb{1}(\cdot)$ is an indicator function taking the value 1 when its argument is true and 0 otherwise. Various ways to decompose a logical property are given in Section 4.1.

While a level-based score function directly correlates logic to score, in many applications the property of interest may not have a suitable notion of levels to exploit; the logical levels may be too coarse or may distribute the probability unevenly. For these cases it is necessary to define a more general score function.

Definition 2. *Let $J_0 \supset J_1 \supset ... \supset J_n$ a set of nested intervals of $\mathbb{R}$ and $\Omega = \Omega_0 \supset \Omega_1 \supset \cdots \supset \Omega_n$ a set of nested subsets of Ω. The mapping $\Phi : \Omega \to \mathbb{R}$ is a* general *score function of property φ if and only if $\forall k : \omega \in \Omega_k \iff \Phi(\omega) \in J_k$ and $\omega \models \varphi \iff \omega \in \Omega_n$ and $\forall i, j \in \{0, \ldots, |\omega|\} : i < j \implies \Phi(\omega_{\leq i}) \leq \Phi(\omega_{\leq j})$*

Informally, Definition 2 states that a general score function requires that the highest scores be assigned to paths that satisfy the overall property and that the score of a path's prefix is non-decreasing with increasing prefix length.

When no formal levels are available, an effective score function may still be defined using heuristics, that only loosely correlate increasing score with increasing probability of satisfying the property. For example, a time bounded property, not explicitly correlated to time, may become increasingly less likely to be satisfied as time runs out (i.e., with increasing path length). The heuristic in this case would assign higher scores to shorter paths. A score function based on coarse logical levels may be improved by using heuristics between the levels.

4.1 Decomposition of a Temporal Logic Formula

Many existing uses of importance splitting employ a natural notion of levels inherent in a specific problem. Systems that do not have an inherent notion of level may be given quasi-natural levels by 'lumping' states of the model into necessarily consecutive states of an abstracted model. This technique is used in the dining philosophers example in Section 6.2.

For the purposes of statistical model checking, it is necessary to link levels to temporal logic. The following subsections describe various ways a logical formula may be decomposed into subformulae that may be used to form a level-based score function. The techniques may be used independently or combined with each other to give the score function greater resolution. Hence, the term 'property' used below refers both to the overall formula and its subformulae.

Since importance splitting depends on successively reaching levels, the initial estimation problem tends to become one of reachability (as in the case of numerical model checking algorithms). We observe from the following subsections that this does not necessarily limit the range of properties that may be considered.

Simple Decomposition. When a property φ is given as an explicit conjunction of n sub-formulae, i.e., $\varphi = \bigwedge_{j=1}^{n} \psi_j$, a simple decomposition into nested properties is obtained by $\varphi_i = \bigwedge_{j=1}^{i} \psi_j, \forall i \in \{1, \ldots, n\}$, with $\varphi_0 \equiv \top$. The associativity

and commutativity of conjunction make it possible to choose an arbitrary order of sub-formulae, with the possibility to choose an order that creates levels with equal conditional probabilities. Properties that are not given as conjunctions may be re-written using DeMorgan's laws in the usual way.

Natural Decomposition. Many rare events are defined with a natural notion of level, i.e., when some quantity in the system reaches a particular value. In physical systems such a quantity might be a distance, a temperature or a number of molecules. In computational systems, the quantity might refer to a loop counter, a number of software objects, or the number of available servers, etc.

Natural levels are thus defined by nested atomic properties of the form $\varphi_i = (l > l_i), \forall i \in \{0, \ldots, n\}$, where l is a state variable, $l_0 = 0 < l_1 < \cdots < l_n$ and $\omega \models \varphi_n \iff l \geq l_n$. When rarity increases with decreasing natural level, the nested properties have the form $\varphi_i = l > l_i, \forall i \in \{0, \ldots, n\}$, with $l_0 = \max(l) > l_1 > \cdots > l_n$, such that $\omega \models \varphi_n \iff l \leq l_n$.

Time may be considered as a natural level if it also happens to be described by a state variable, however in the following subsection it is considered in terms of the bound of a temporal operator.

Decomposition of Temporal Operators. The following Propositions hold:

1. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\mathbf{F}^{\leq t}\varphi_n \Rightarrow \mathbf{F}^{\leq t}\varphi_{n-1})$
2. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\mathbf{G}^{\leq t}\varphi_n \Rightarrow \mathbf{G}^{\leq t}\varphi_{n-1})$
3. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\mathbf{X}\varphi_n \Rightarrow \mathbf{X}\varphi_{n-1})$
4. $(\varphi_n \Rightarrow \varphi_{n-1} \wedge \psi_m \Rightarrow \psi_{m-1}) \Longrightarrow (\varphi_n \mathbf{U} \psi_m \Rightarrow \varphi_{n-1} \mathbf{U} \psi_{m-1})$
5. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_n \Rightarrow \mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_{n-1})$
6. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\forall \omega \models \mathbf{G}^{\leq t}\varphi_n : \exists t' \geq t \mid \omega \models \mathbf{G}^{\leq t'}\varphi_{n-1})$
7. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\forall \omega \models \mathbf{F}^{\leq t}\varphi_n : \exists t' \leq t \mid \omega \models \mathbf{F}^{\leq t'}\varphi_{n-1})$
8. $(t' \geq t) \Longrightarrow (\mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_n \Rightarrow \mathbf{F}^{\leq t'}\mathbf{G}^{\leq s}\varphi_n)$
9. $(s' \leq s) \Longrightarrow (\mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_n \Rightarrow \mathbf{F}^{\leq t}\mathbf{G}^{\leq s'}\varphi_n)$
10. $(t' \geq t \wedge s' \leq s) \Longrightarrow (\mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_n \Rightarrow \mathbf{F}^{\leq t'}\mathbf{G}^{\leq s'}\varphi_n)$
11. $(\varphi_n \Rightarrow \varphi_{n-1}) \Longrightarrow (\forall \omega \models \mathbf{F}^{\leq t}\mathbf{G}^{\leq s}\varphi_n : \exists t' \leq t \wedge s' \geq s \mid \omega \models \mathbf{F}^{\leq t'}\mathbf{G}^{\leq s'}\varphi_{n-1})$

Temporal Decomposition. From Proposition 6, properties having the form $\varphi = \mathbf{G}^t\psi$ may be decomposed in terms of t. For an arbitrary suffix $\omega_{\geq k} = s_k \xrightarrow{t_k} s_{k+1} \xrightarrow{t_{k+1}} s_{k+2} \xrightarrow{t_{k+2}} \cdots$, we have $(\omega_{\geq k} \models \mathbf{G}^t\psi) \leftrightarrow (\omega_{\geq k} \models \psi) \wedge (\omega_{\geq k+1} \models \psi) \wedge \cdots \wedge (\omega_{k+m} \models \psi)$, for some m such that $\sum_{j=k}^{m+k} t_j \leq t \ \wedge \ \sum_{j=k}^{m+k+1} t_j > t$. This has the form required for a simple decomposition, giving nested properties of the form $\varphi_i = \mathbf{G}^{l_i}\psi, \forall i \in \{1, \ldots, n\}$, where $l_1 = 0 < l_2 < \cdots < l_n = t$, with $\varphi_0 \equiv \top$.

Properties having the form $\varphi = \mathbf{F}^t\psi$ evaluate to disjunctions in terms of time. From Proposition 7, it is plausible to construct nested properties of the form $\varphi_i = \mathbf{F}^{t+l_i}\psi, \forall i \in \{1, \ldots, n\}$, with $l_1 > l_2 > \cdots > l_n = 0$ and $\varphi_0 \equiv \top$. Some caution is required if t is the value given in the overall property. If trace ω satisfies $\mathbf{F}^{t'}$ but not $\mathbf{F}^t$, any prefix of ω does not satisfy $\mathbf{F}^t$. The requirement for $\mathbf{F}^{t'}$ to have a lower score than $\mathbf{F}^t$ conflicts with the requirement of a score function $\forall i, j \in \{0, \ldots, |\omega|\} : i < j \implies \Phi(\omega_{\leq i}) \leq \Phi(\omega_{\leq j})$.

Heuristic Decomposition. The decomposition of a property into logical levels may not necessarily result in an adequate score function: there may be insufficient levels, the levels may be irrelevant to the overall property or the levels may not evenly distribute the probability. In such cases it may be desirable to define intermediate levels based on heuristics – approximate correlations between a path and its probability to satisfy the property. For example, $\varphi_i = \mathbf{F}^{t+l_i}\psi$ may not form legitimate nested properties with positive l_i, but may nevertheless be used as a heuristic with $l_i \in [-t, 0]$.

Note that a heuristic score function that respects Definition 2 will give an unbiased estimate when used with an unbiased importance splitting algorithm. The effectiveness of a heuristic is dependent on how well it correlates path prefixes with the probability of eventually satisfying the overall property.

5 Importance Splitting Algorithms

We give two importance splitting pseudo-algorithms; one with fixed levels defined a priori and one that finds optimal levels adaptively. N denotes the number of simulations performed at each level. Levels, denoted τ, are defined as values of score function $\Phi(\omega)$, where ω is a path. τ_k is the k^{th} level and ω_i^k is the i^{th} simulation on level k. $\tilde{\gamma}_k$ is the estimate of γ_k, the k^{th} conditional probability $\mathrm{P}(\Phi(\omega) \geq \tau_k \mid \Phi(\omega) \geq \tau_{k-1})$.

5.1 Fixed Level Algorithm

The fixed level algorithm follows from the general description given in Section 3.2. Its advantages are that it is simple, it has low computational overhead and the resulting estimate is unbiased. Its disadvantage is that the levels must often be guessed by trial and error – adding to the overall computational cost.

In Algorithm 1, $\tilde{\gamma}$ is an unbiased estimate (see, e.g., [7]). Furthermore, from Proposition 3 in [3], we can deduce the following $(1-\alpha)$ confidence interval:

$$CI = \left[\tilde{\gamma}\left(\frac{1}{1+\frac{z_\alpha \sigma}{\sqrt{N}}}\right), \tilde{\gamma}\left(\frac{1}{1-\frac{z_\alpha \sigma}{\sqrt{N}}}\right)\right] \qquad \text{with} \qquad \sigma^2 \geq \sum_{k=1}^{M} \frac{1-\gamma_k}{\gamma_k}, \quad (5)$$

where z_α is the $1-\frac{\alpha}{2}$ quantile of the standard normal distribution. Hence, with confidence $100(1-\alpha)\%$, $\gamma \in CI$. σ is reduced by making all γ_k equal and large. For given γ, this implies increasing M, further motivating fine grained score functions. When it is not possible to define γ_k arbitrarily, the confidence interval may nevertheless be reduced by increasing N. The inequality for σ arises because the independence of initial states diminishes with increasing levels: unsuccessful traces are discarded and new initial states are drawn from successful traces. Several possibilities have been provided in [3] to minimise this dependence effect. For the sake of simplicity, in the following we assume that this goal is achieved, such that CI is calcultaed with σ estimated by the square root of $\sum_{k=1}^{M} \frac{1-\tilde{\gamma_k}}{\tilde{\gamma_k}}$.

Algorithm 1. Fixed levels

Let $(\tau_k)_{1 \leq k \leq M}$ be the sequence of thresholds
Let *stop* be a termination condition
$\forall j \in \{1, \ldots, N\}$, set $\tilde{\omega}_j^1 = \emptyset$
for $1 \leq k \leq M$ **do**
  $\forall j \in \{1, \ldots, N\}$, using prefix $\tilde{\omega}_j^k$, generate path ω_j^k until $(\Phi(\omega_j^k) \geq \tau_k) \vee stop$
  $I_k = \{\forall j \in \{1, \ldots, N\} : \Phi(\omega_j^k) \geq \tau_k\}$
  $\tilde{\gamma}_k = \frac{|I_k|}{N}$
  $\forall j \in I_k$, $\tilde{\omega}_j^{k+1} = \omega_j^k$
  $\forall j \notin I_k$, let $\tilde{\omega}_j^{k+1}$ be a copy of ω_i^k with $i \in I_k$ chosen uniformly randomly
$\tilde{\gamma} = \prod_{k=1}^{M} \tilde{\gamma}_k$

5.2 Adaptive Level Algorithm

The cost of finding good levels must be included in the overall computational cost of importance splitting. An alternative to trial and error is to use an adaptive level algorithm that discovers its own optimal levels.

Algorithm 2. Adaptive levels

Let $\tau_\varphi = \min\{\Phi(\omega) \mid \omega \models \varphi\}$ be the minimum score of paths that satisfy φ
Let N_k be the pre-defined number of paths to keep per iteration
$k = 1$
$\forall j \in \{1, \ldots, N\}$, generate path ω_j^k
repeat
  Let $T = \{\Phi(\omega_j^k), \forall j \in \{1, \ldots, N\}\}$
  Find minimum $\tau_k \in T$ such that $|\{\tau \in T : \tau > \tau_k\}| \geq N_k$
  $\tau_k = \min(\tau_k, \tau_\varphi)$
  $I_k = \{j \in \{1, \ldots, N\} : \Phi(\omega_j^k) > \tau_k\}$
  $\tilde{\gamma}_k = \frac{|I_k|}{N}$
  $\forall j \in I_k$, $\omega_j^{k+1} = \omega_j^k$
  for $j \notin I_k$ **do**
    choose uniformly randomly $l \in I_k$
    $\tilde{\omega}_j^{k+1} = \max_{|\omega|} \{\omega \in pref(\omega_l^k) : \Phi(\omega) < \tau_k\}$
    generate path ω_j^{k+1} with prefix $\tilde{\omega}_j^{k+1}$
  $M = k$
  $k = k + 1$
until $\tau_k > \tau_\varphi$;
$\tilde{\gamma} = \prod_{k=1}^{M} \tilde{\gamma}_k$

Algorithm 2 is an adaptive level importance splitting algorithm based on [4]. It works by pre-defining a fixed number N_k of simulation traces to retain at each level. With the exception of the last level, the conditional probability of each level is then nominally N_k/N.

Use of the adaptive algorithm may lead to gains in efficiency (no trial and error, reduced overall variance), however the final estimate has a bias of order $\frac{1}{N}$, i.e., $\mathrm{E}(\tilde{\gamma}) = \gamma + \mathcal{O}(N^{-1})$. The overestimation (potentially not a problem when estimating rare critical failures) is negligible with respect to σ, such that the confidence interval remains that of the fixed level algorithm. Furthermore, under some regularity conditions, the bias can be asymptotically corrected. The estimate of γ has the form $r_0{\gamma_0}^{M_0}$, with $M_0 = M-1$, $r_0 = \gamma{\gamma_0}^{-M_0}$ and $\frac{E[\tilde{\gamma}]-\gamma}{\gamma} \sim \frac{M_0}{N}\frac{1-\gamma_0}{\gamma_0}$ when N goes to infinity. Using the expansion

$$\tilde{\gamma} = \gamma\left(1 + \frac{1}{\sqrt{N}}\sqrt{M_0\frac{1-\gamma_0}{\gamma_0} + \frac{1-r_0}{r_0}}Z + \frac{1}{N}M_0\frac{1-\gamma_0}{\gamma_0} + o\left(\frac{1}{N}\right)\right),$$

with Z a standard normal variable, $\tilde{\gamma}$ is corrected by dividing it by $1 + \frac{M_0(1-\gamma_0)}{N\gamma_0}$. See [3] for more details.

6 Examples

We have adapted models from the literature to illustrate the use of importance splitting with statistical model checking. All simulations were performed using our statistical model checking platform PLASMA in which the previous algorithms have been implemented [11].

6.1 Biochemical Network

The network of chemical reactions given below is typical of biochemical systems and demonstrates the potential of SMC to handle the enormous state spaces of biological models.

We consider a well stirred chemically reacting system comprising five reactants (molecules of type A, B, C, D, E), a dimerisation reaction (6) and two decay reactions (7,8).

$$A + B \xrightarrow{1} C \tag{6}$$

$$C \xrightarrow{1} D \tag{7}$$

$$D \xrightarrow{1} E \tag{8}$$

The semantics of (6) is that if a molecule of type A encounters a molecule of type B they will combine to form a molecule of type C after a delay drawn from an exponential distribution with mean 1. The decay reactions have the semantics that a molecule of type C (D) spontaneously decays to a molecule of type D (E) after a delay drawn from an exponential distribution with mean 1. A typical simulation run is illustrated in Figure 1. A and B combine rapidly to form C, that peaks before decaying slowly to D. The production of D also peaks, while E rises monotonically.

With an initial vector of molecules $(1000, 1000, 0, 0, 0)$, corresponding to types (A, B, C, D, E), the total number of states is less than 10^9, but beyond the current practical capability of exhaustive probabilistic model checking. It is possible for the number of molecules of D to reach 1000, however $D > 400$ is unusual. We thus define a suitably rare property to be $\varphi = \mathbf{F}^t D > 460$, with t initially 3000 steps, chosen to be adequately long. To apply Algorithm 1, we set $N = 1000$ and

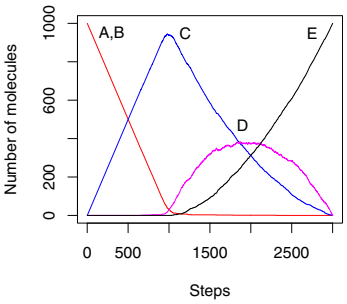


Fig. 1. A typical stochastic simulation trace of reactions (6-8)

Table 1. Chemical network conditional probability estimates based on 1000 runs of Algorithm 1 using $N = 1000$. $\sigma_{estimator}$ is estimated using the sample means.

Probability	Estimate	$\sigma_{estimator}$
$\mathrm{P}(D > 390)$	0.182	0.012
$\mathrm{P}(D > 400 \mid D > 390)$	0.299	0.021
$\mathrm{P}(D > 410 \mid D > 400)$	0.201	0.019
$\mathrm{P}(D > 420 \mid D > 410)$	0.134	0.017
$\mathrm{P}(D > 430 \mid D > 420)$	0.088	0.016
$\mathrm{P}(D > 440 \mid D > 430)$	0.057	0.015
$\mathrm{P}(D > 450 \mid D > 440)$	0.035	0.012
$\mathrm{P}(D > 460 \mid D > 450)$	0.021	0.009
$\mathrm{P}(D > 460)$	8.1×10^{-9}	1.29×10^{-8}

define a nested sequence of properties $\varphi_0 = \top$, $\varphi_i = \mathbf{F}^t D \geq \tau_i$, with $\tau_1 = 390$, $\tau_2 = 400$, $\tau_3 = 410$, $\tau_4 = 420$, $\tau_5 = 430$, $\tau_6 = 440$, $\tau_7 = 450$ and $\tau_8 = 460$. The score function is thus a mapping from paths to τ. τ_1 was found by trial and error, chosen to produce sufficient occurrences of the property on the first level. The other values are equally spaced.

We executed the algorithm 1000 times using the parameters given above. The results are given in Table 1. The standard deviation of the estimator, $\sigma_{estimator}$, is estimated in each case using the sample mean. An individual estimate is achieved with 8000 simulation runs; approx. 1.5×10^4 times fewer than the expected number to see a single instance of the rare property.

Algorithm 1 estimates $\mathrm{P}(D > 460) \approx 8.1 \times 10^{-9}$ with 8 levels, implying an optimal (to minimise variance) per-level conditional probability of approx. 0.097. Based on 100 executions, with $N = 1000$ and N_k thus set to 97, Algorithm 2 chose average levels $\hat{\tau}_1 = 396.0$, $\hat{\tau}_2 = 414.5$, $\hat{\tau}_3 = 426.3$, $\hat{\tau}_4 = 434.6$, $\hat{\tau}_5 = 441.8$, $\hat{\tau}_6 = 448.3$, $\hat{\tau}_7 = 454.1$ and $\hat{\tau}_8 = 459.0$. There is apparently some scope with this score function to increase the number of levels and thus increase the confidence of the estimate according to (5). This is left to a future investigation.

To compare the estimates of Algorithm 2 and Algorithm 1, we set $N = 1000$ and $N_k = 100$, giving a nominal conditional probability of 0.1 per level. The average levels chosen by Algorithm 2 under these circumstances were $\hat{\tau}_1 = 395.8$, $\hat{\tau}_2 = 414.0$, $\hat{\tau}_3 = 425.4$, $\hat{\tau}_4 = 433.7$, $\hat{\tau}_5 = 440.8$, $\hat{\tau}_6 = 447.3$, $\hat{\tau}_7 = 453.1$ and $\hat{\tau}_8 = 458.2$. These levels have fractionally closer spacing than those with $N_k = 97$, reflecting the marginally increased nominal per-level probability. With 1000 executions, Algorithm 2 estimates $\mathrm{P}(D > 460) \approx 1.4 \times 10^{-8}$, compared to the estimate of 8.1×10^{-9} with Algorithm 1. Given the estimated standard deviation of the fixed level estimator, this empirical difference is ascribed to statistical variance rather than the overestimate predicted by theory. Furthermore, a direct computation shows that the estimate of Algorithm 2 is within the 95% confidence interval of Algorithm 1 ($CI = [5 \times 10^{-9}; 2.4 \times 10^{-8}]$).

6.2 Dining Philosophers

We construct a rare event based on the well known probabilistic solution [16] of Dijkstra's dining philosophers problem [8]. In this example, there are no natural counters to exploit, so levels must be constructed by considering 'lumped' states.

A number of philosophers sit at a circular table with an equal number of chopsticks; a chopstick being placed within reach of two adjacent philosophers. Philosophers think and occasionally wish to eat from a communal bowl. To eat, a philosopher must independently pick up two chopsticks: one from the left and one from the right. Having eaten, the philosopher replaces the chopsticks and returns to thinking. A problem of concurrency arises because a philosopher's neighbour(s) may have already taken the chopstick(s). Lehmann and Rabin's solution [16] is to allow the philosophers to make probabilistic choices.

We consider a model of 100 'free' philosophers [16]. The number of states in the model is approx. 10^{95}; 10^{15} times more than the estimated number of protons in the universe. The possible states of an individual philosopher can be abstracted to those shown in Fig. 2.

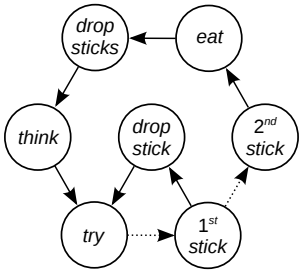


Fig. 2. An abstract model of a dining philosopher

Table 2. Dining philosophers conditional probability estimates based on 100 runs of Algorithm 1 with $N = 1000$. $\sigma_{estimator}$ is estimated using the sample means.

Probability	Estimate	$\sigma_{estimator}$
$\mathrm{P}(\mathit{try})$	0.055	0.007
$\mathrm{P}(1^{\mathsf{st}}\mathit{stick}\vert\mathit{try})$	0.029	0.006
$\mathrm{P}(2^{\mathsf{nd}}\mathit{stick}\vert 1^{\mathsf{st}}\mathit{stick})$	0.017	0.005
$\mathrm{P}(\mathit{eat}\vert 2^{\mathsf{nd}}\mathit{stick})$	0.010	0.005
$\mathrm{P}(\mathit{drop\ sticks}\vert\mathit{eat})$	0.005	0.004
$\mathrm{P}(\mathit{drop\ sticks})$	1.7×10^{-9}	1.95×10^{-9}

Thinking is the initial state of all philosophers. The transitions denoted by dotted lines in Figure 2 are dependent on the availability of chopsticks. All transitions are controlled by stochastic rates and made in competition with the transitions of other philosophers. With increasing numbers of philosophers, it is increasingly unlikely that a specific philosopher will be satisfied (i.e., that the philosopher will reach the state *drop sticks*) within a given number of steps from the initial state. We thus define a rare property $\varphi = \mathbf{F}^t \mathit{drop\ sticks}$, with t initially 7, denoting the property that a given philosopher will reach state *drop sticks* within 7 steps. Thus, using the states of the abstract model, we decompose φ into nested properties $\varphi_0 = \top$, $\varphi_1 = \mathbf{F}^t \mathit{try}$, $\varphi_2 = \mathbf{F}^t 1^{\mathsf{st}}\mathit{stick}$, $\varphi_3 = \mathbf{F}^t 2^{\mathsf{nd}}\mathit{stick}$, $\varphi_4 = \mathbf{F}^t \mathit{eat}$ and $\varphi_5 = \mathbf{F}^t \mathit{drop\ sticks}$. The score function, not used explicitly here, is that defined by (4).

We executed Algorithm 1 100 times and obtained the results given in Table 2. The final estimate is achieved with approx. 10^5 fewer simulations than would be expected to see a single occurrence of the property using simple Monte Carlo.

6.3 Repair Model

We consider a repair model from the rare event literature (Ex. 1 in [19]), which represents a class of systems that is known to be challenging for parametrised importance sampling; the use of 'group repair' causes them to be 'unbalanced' [19] and renders simple biasing schemes unable to bound the relative error [21].

The model comprises three types of components, with n components per type, that may fail and be repaired at certain probabilistic rates. Each type of component has a different rate of failing and components fail independently. The initial state has no failed components. Repairs are prioritised: components of type 1 are repaired before those of type 2 and type 2 are repaired before type 3. There is a common repair rate, but types 1 and 2 are repaired in groups (all failed components are repaired in one event) while type 3 are repaired singly.

We consider the *total failure entrance probability* (the probability that all components fail, without the system returning to the initial state) expressed as $\gamma = \mathrm{P}(\omega \models \mathit{init} \wedge \mathbf{X}(\neg \mathit{init}\mathbf{U}^t \mathit{failure}))$, with t infinite. Let fail_1, fail_2 and fail_3 denote the instantaneous number of failed components of types 1, 2 and 3, respectively, then *init* is defined as $\mathit{fail}_1 = 0 \wedge \mathit{fail}_2 = 0 \wedge \mathit{fail}_3 = 0$ and *failure* is defined as $\mathit{fail}_1 = n \wedge \mathit{fail}_2 = n \wedge \mathit{fail}_3 = n$. We set $n = 4$ to create a model with a rare event that is nevertheless tractable to numerical analysis. We thus find that $\gamma = 1.177 \times 10^{-7}$ to four significant figures.

The property $\varphi = \mathit{init} \wedge \mathbf{X}(\neg \mathit{init}\mathbf{U}^t \mathit{failure})$ has the form of a conjunction, but a simple decomposition is trivial. Using Proposition 3 we can decompose $\mathbf{X}$ and using Proposition 4 we can decompose $\mathbf{U}$. *init* is a conjunction, but is used negated so can not be usefully decomposed. *failure* can be decomposed as a simple conjunction or in terms of natural levels of failed components. We combine these and consider nested properties based on the total number of failed components $\mathit{totalfail} = \mathit{fail}_1 + \mathit{fail}_2 + \mathit{fail}_3$. The score function is then just a mapping from paths to *totalfail*.

We thus define levels $\tau_0 = 0$, $\tau_1 = 2, \ldots, \tau_i = i+1, \ldots, \tau_{11} = 12$ and construct nested properties of the form $\varphi_i = \mathit{init} \wedge \mathbf{X}(\neg \mathit{init}\mathbf{U}^t \mathit{totalfail} \geq \tau_i)$. We applied Algorithm 1 100 times and achieved the results shown in Table 3. Using the numerical model checker PRISM[1] to calculate the true probabilities, we calculate the standard deviations of our estimators ($\sigma_{estimator}$). We conclude that we are able to accurately estimate γ with approx. 800 fewer simulations than would be expected to produce a single example of the rare property.

The results are illustrated in Fig. 3, where the inset *box and whisker* plot shows the overall performance of the importance splitting estimator with respect to the true value of γ. The use of a logarithmic scale serves to demonstrate how the relative error increases with decreasing estimated probability, motivating the need to find optimal levels. Given the infinite time horizon of the property in this example, we hypothesise that it might be possible to use temporal decomposition to increase the granularity of the score function and thus balance the conditional probabilities of the levels. This is left to future work.

[1] `www.prismmodelchecker.org`

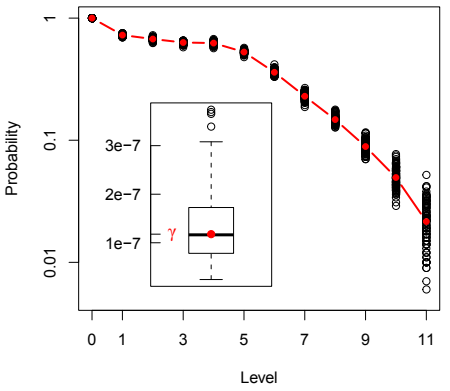


Fig. 3. Estimated (black) and true (red) conditional probabilities for repair model (line only to guide the eye). Inset, overall estimate (black line) and true value (red dot).

Table 3. Estimated conditional and overall probabilities for repair model, based on 100 runs of Algorithm 1 with $N = 1000$. $\sigma_{estimator}$ is calculated w.r.t. the true values.

Probability	Estimate	$\sigma_{estimator}$
$\mathrm{P}(\varphi_1 \mid \varphi_0)$	0.725	0.015
$\mathrm{P}(\varphi_2 \mid \varphi_1)$	0.673	0.016
$\mathrm{P}(\varphi_3 \mid \varphi_2)$	0.628	0.015
$\mathrm{P}(\varphi_4 \mid \varphi_3)$	0.622	0.019
$\mathrm{P}(\varphi_5 \mid \varphi_4)$	0.529	0.015
$\mathrm{P}(\varphi_6 \mid \varphi_5)$	0.360	0.017
$\mathrm{P}(\varphi_7 \mid \varphi_6)$	0.231	0.015
$\mathrm{P}(\varphi_8 \mid \varphi_7)$	0.149	0.011
$\mathrm{P}(\varphi_9 \mid \varphi_8)$	0.091	0.010
$\mathrm{P}(\varphi_{10} \mid \varphi_9)$	0.050	0.010
$\mathrm{P}(\varphi_{11} \mid \varphi_{10})$	0.023	0.009
$\mathrm{P}(\omega \models \varphi_{11})$	1.34×10^{-7}	8.12×10^{-8}

7 Conclusion

We have introduced the notion of using importance splitting with statistical model checking to verify rare properties. We have described how such properties must be decomposed to facilitate importance splitting and have demonstrated the procedures on several examples. We have described two importance splitting algorithms that may be constrained to give results within confidence bounds. Overall, we have shown that the application of importance splitting to statistical model checking has great potential.

References

1. Baier, C., Katoen, J.-P.: Principles of Model Checking. Representation and Mind Series. MIT Press (2008)
2. Barbot, B., Haddad, S., Picaronny, C.: Coupling and importance sampling for statistical model checking. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 331–346. Springer, Heidelberg (2012)
3. Cérou, F., Del Moral, P., Furon, T., Guyader, A.: Sequential Monte Carlo for rare event estimation. Statistics and Computing 22, 795–808 (2012)
4. Cérou, F., Guyader, A.: Adaptive multilevel splitting for rare event analysis. Stochastic Analysis and Applications 25, 417–443 (2007)
5. Chernoff, H.: A Measure of Asymptotic Efficiency for Tests of a Hypothesis Based on the sum of Observations. Ann. Math. Statist. 23(4), 493–507 (1952)
6. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge (1999)
7. Del Moral, P.: Feynman-Kac Formulae: Genealogical and Interacting Particle Systems with Applications. Probability and Its Applications. Springer (2004)

8. Dijkstra, E.W.: Hierarchical ordering of sequential processes. Acta Informatica 1, 115–138 (1971)
9. Gillespie, D.T.: Exact stochastic simulation of coupled chemical reactions. Journal of Physical Chemistry 81, 2340–2361 (1977)
10. Glasserman, P., Heidelberger, P., Shahabuddin, P., Zajic, T.: Multilevel splitting for estimating rare event probabilities. Oper. Res. 47(4), 585–600 (1999)
11. Jegourel, C., Legay, A., Sedwards, S.: A Platform for High Performance Statistical Model Checking – PLASMA. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 498–503. Springer, Heidelberg (2012)
12. Jegourel, C., Legay, A., Sedwards, S.: Cross-Entropy Optimisation of Importance Sampling Parameters for Statistical Model Checking. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 327–342. Springer, Heidelberg (2012)
13. Kahn, H.: Random sampling (Monte Carlo) techniques in neutron attenuation problems. Nucleonics 6(5), 27 (1950)
14. Kahn, H., Harris, T.E.: Estimation of Particle Transmission by Random Sampling. In: Applied Mathematics. series 12, vol. 5, National Bureau of Standards (1951)
15. Kahn, H., Marshall, A.W.: Methods of Reducing Sample Size in Monte Carlo Computations. Operations Research 1(5), 263–278 (1953)
16. Lehmann, D., Rabin, M.O.: On the Advantage of Free Choice: A Symmetric and Fully Distributed Solution to the Dining Philosophers Problem (Extended Abstract). In: Proc. 8th Ann. Symposium on Principles of Programming Languages, pp. 133–138 (1981)
17. Metropolis, N., Ulam, S.: The Monte Carlo Method. Journal of the American Statistical Association 44(247), 335–341 (1949)
18. Reijsbergen, D., de Boer, P.-T., Scheinhardt, W., Haverkort, B.: Rare event simulation for highly dependable systems with fast repairs. Performance Evaluation 69(7-8), 336–355 (2012)
19. Ridder, A.: Importance sampling simulations of markovian reliability systems using cross-entropy. Annals of Operations Research 134, 119–136 (2005)
20. Rosenbluth, M.N., Rosenbluth, A.W.: Monte Carlo Calculation of the Average Extension of Molecular Chains. Journal of Chemical Physics 23(2) (February 1955)
21. Shahabuddin, P.: Importance Sampling for the Simulation of Highly Reliable Markovian Systems. Management Science 40(3), 333–352 (1994)
22. Villén-Altamirano, M., Villén-Altamirano, J.: RESTART: A Method for Accelerating Rare Event Simulations. In: Cohen, J.W., Pack, C.D. (eds.) Queueing, Performance and Control in ATM, pp. 71–76. Elsevier (1991)
23. Wald, A.: Sequential Tests of Statistical Hypotheses. The Annals of Mathematical Statistics 16(2), 117–186 (1945)

Minimal Sets over Monotone Predicates in Boolean Formulae

Joao Marques-Silva[1,2], Mikoláš Janota[2], and Anton Belov[1,⋆]

[1] CASL, University College Dublin, Ireland
[2] IST/INESC-ID, Technical University of Lisbon, Portugal

Abstract. The importance and impact of the Boolean satisfiability (SAT) problem in many practical settings is well-known. Besides SAT, a number of computational problems related with Boolean formulas find a wide range of practical applications. Concrete examples for CNF formulas include computing prime implicates (PIs), minimal models (MMs), minimal unsatisfiable subsets (MUSes), minimal equivalent subsets (MESes) and minimal correction subsets (MCSes), among several others. This paper builds on earlier work by Bradley and Manna and shows that all these computational problems can be viewed as computing a minimal set subject to a monotone predicate, i.e. the MSMP problem. Thus, if cast as instances of the MSMP problem, these computational problems can be solved with the same algorithms. More importantly, the insights provided by this result allow developing a new algorithm for the general MSMP problem, that is asymptotically optimal. Moreover, in contrast with other asymptotically optimal algorithms, the new algorithm performs competitively in practice. The paper carries out a comprehensive experimental evaluation of the new algorithm on the MUS problem, and demonstrates that it outperforms state of the art MUS extraction algorithms.

1 Introduction

The theoretical and practical significance of Boolean Satisfiability (SAT) cannot be overstated. This is illustrated by the ever increasing number of practical applications of SAT solvers (see [23,16] for recent overviews). Besides SAT, other computational problems related with Boolean formulas are of interest, both from theoretical and practical perspectives. These include computing prime implicates (PIs) (given an original implicate), minimal models (MMs), minimal unsatisfiable subsets (MUSes), minimal equivalent (or irredundant) subsets (MESes), and minimal correction subsets (MCSes), among several others. Some of these problems find applications in verification. For example, prime implicates have

⋆ This work is partially supported by SFI grant BEACON (09/IN.1/I2618), by FCT grants ATTEST (CMU-PT/ELE/0009/2009) and POLARIS (PTDC/EIA-CCO/123051/2010), and by INESC-ID's multiannual PIDDAC funding PEst-OE/EEI/LA0021/2013.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 592–607, 2013.

been used in inductive strengthening [8,9], whereas MUSes and MCSes find application in proof-based abstraction [26] and counterexample-guided abstraction refinement [1]. Besides verification, the range of applications include knowledge representation [12], non-monotonic reasoning [25], and description logics [30].

Over the years, many different algorithms have been proposed for the above computational problems. Thus, there are dedicated algorithms for computing PIs [8,9,24], MUSes [5,17], MCSes [2,21,15,27], among others. The approach for computing PIs proposed in [8,9] is described using a general framework for computing a minimal set subject to a monotone predicate. We refer to this problem as the *minimal set over a monotone predicate* (MSMP) problem. This paper exploits this insight [8,9], and shows that all of the above computational problems (i.e. PIs, MMs, MUSes, MCSes, etc.) can be represented as instantiations of the MSMP problem. This result immediately implies that algorithms for MSMP can be used for solving *any* of the computational problems listed above, including PIs, MMs, MUSes, MCSes, among others. In addition, existing algorithms for any of these computational problems can be abstracted to the general framework of the MSMP problem. More importantly, the insights provided by these observations allow developing a new algorithm for the MSMP, that is in some sense optimal (i.e. it is asymptotically as efficient as the most efficient algorithms), but that performs well in practice (in contrast with other optimal algorithms developed for specific computational problems, e.g. PIs and MUSes [19,8,9]). This new algorithm is specialized for the case of MUS extraction, thus illustrating how existing pruning techniques (and new ones) can be integrated into the general MSMP algorithm. Experimental results, obtained on well-known practically-relevant problem instances, demonstrate that the new algorithm outperforms the current state of the art MUS extraction algorithm [5,6].

The paper is organized as follows. The next section introduces the definitions used throughout the paper. Section 4 presents the MSMP problem, and shows that the computational problems listed above can be formulated as instantiations of the MSMP problem. Afterwards, Section 5 develops a new algorithm for the MSMP problem (and so it is applicable to PIs, MMs, MUSes, MCSes, etc.). Section 6 presents and analyzes the experimental results for the case of MUS extraction. Finally, Section 7 concludes the paper.

2 Preliminaries

This section briefly introduces the definitions used throughout. Additional standard definitions can be found elsewhere (e.g. [16,23,7]). Boolean formulas are represented in calligraphic font, $\mathcal{F}$, $\mathcal{M}$, $\mathcal{U}$, $\mathcal{T}$, etc. A Boolean formula in conjunctive normal form (CNF) is defined as a finite set of finite sets of literals. Where appropriate, a CNF formula will also be understood as a conjunction of disjunctions of literals. The variables of formula $\mathcal{F}$ are denoted by $\mathsf{var}(\mathcal{F})$. An assignment is a map $\mu : \mathsf{var}(\mathcal{F}) \rightarrow \{0, 1\}$. A clause is satisfied by an assignment if one of its literals is assigned value 1. A model of $\mathcal{F}$ is an assignment that satisfies all clauses in $\mathcal{F}$.

When manipulating CNF formulas, it will often be necessary to consider the clauses in a given range. Given CNF formula $\mathcal{F}$ and range $i..j$, where $\mathcal{F}$ is viewed as a sequence $\langle c_1, c_2, \ldots, c_{|\mathcal{F}|}\rangle$, the notation $\mathcal{F}_{i..j}$ represents the clauses in the range $i..j$, i.e. $\{c_i, c_{i+1}, \ldots, c_j\}$ for $j \geq i$, or $\emptyset$ if $j < i$. This same notation will be used for other sets. The following definitions will be used throughout [10,8,21,5].

Definition 1 (Prime Implicate given Implicate). *A prime implicate π of $\mathcal{F}$ given an implicate c is a minimal subset of the literals in c such that $\mathcal{F} \models \pi$.*

Definition 2 (Minimal Model). *A minimal model is a model μ of $\mathcal{F}$ such that the set of true variables is minimal with respect to set containment.*

Definition 3 (MU). *$\mathcal{F}$ is* Minimally Unsatisfiable *(MU) iff $\mathcal{F}$ is unsatisfiable and $\forall_{c\in\mathcal{F}}, \mathcal{F} \setminus \{c\}$ is satisfiable.*

Definition 4 (MUS). *$\mathcal{M}$ is a* Minimally Unsatisfiable Subformula *(MUS) of $\mathcal{F}$ iff $\mathcal{M} \subseteq \mathcal{F}$ and $\mathcal{M}$ is minimally unsatisfiable.*

Definition 5 (MCS). *$\mathcal{C} \subseteq \mathcal{F}$ is a* Minimal Correction Subset *(MCS) iff $\mathcal{F} \setminus \mathcal{C}$ is satisfiable and $\forall_{c\in\mathcal{C}}, \mathcal{F} \setminus (\mathcal{C} \setminus \{c\})$ is unsatisfiable.*

The definition of other computational problems, mentioned in the paper but not explicitly addressed, can be found in the references [21,5,4]. Finally, although the paper focuses on CNF formulas, the results can be extended to disjunctive normal form (DNF) formulas, provided the computational problems of interest are modified accordingly.

3 Related Work

This section overviews work on PIs (subject to an implicate), MMs, MUSes, and MCSes. There is a large body of work on computing prime implicates, e.g. see [24] for an overview. However, the problem this paper addresses focuses on computing a prime implicate starting from an implicate. This problem is studied in [8,9] (also see references therein). A key insight of Bradley&Manna's work is in representing the problem of computing a prime implicate in terms of computing a minimal set subject to a monotone predicate. This insight is extensively used in our work. Another contribution of [8,9] is an optimal (when there exists a single unique minimal set) algorithm for computing a prime implicate. As highlighted later, this optimal algorithm for computing a prime implicate corresponds to the QuickXPlain algorithm for MUS extraction [19]. Minimal models find a wide range of applications in Artificial Intelligence. A concrete example is non-monotonic reasoning [25]. A recent example of applying minimal models is [31].

Recent years have seen a large body of work on computing MUSes (see [5,17] and references therein). A wealth of algorithms have been proposed, of which the most efficient in practice is the so-called *hybrid* algorithm [22,5]. Essential to modern algorithms are techniques to reduce the number of calls to a SAT oracle. The most effective are clause set refinement [14,22] and model rotation [22,5].

A theoretically optimal algorithm for MUS extraction is the QuickXPlain algorithm [19], which in practice performs poorly on CNF formulas. MCSes find a large number of applications, an example of which is in using hitting set duality for enumerating MUSes [28,21]. Recent algorithms for computing MCSes include the use of Maximum Satisfiability [21], iterative clause analysis [27], and a modified QuickXPlain algorithm [15].

To our best knowledge, there is no work relating these computational problems (and the ones mentioned in Section 1), and showing that all can be solved with the same algorithms. This is the focus of the next section.

4 Minimal Sets over Monotone Predicates

This section introduces the minimal model subject to a monotone predicate (MSMP) problem, using the framework developed in [8,9], and shows that several computational problems on CNF formulas can be mapped to the MSMP problem. This section also illustrates how some of the existing MUS extraction algorithms can be adapted to the MSMP problem. Finally, the section analyzes how well-known pruning techniques used in MUS extraction can be used in algorithms for the MSMP problem.

4.1 The General Framework

This section revisits the approach presented in [8,9]. Let $\mathcal{F}$ be a CNF formula. Moreover, let $\mathcal{R}$ be a set of elements (in some way related with $\mathcal{F}$), i.e. the *reference set*. A predicate $p : 2^{\mathcal{R}} \rightarrow \{0, 1\}$, defined on $\mathcal{R}$, is said to be *monotone* if it has the following properties:

1. $p(\mathcal{R})$ holds.
2. If $p(\mathcal{R}_0)$ holds, and $\mathcal{R}_0 \subseteq \mathcal{R}_1 \subseteq \mathcal{R}$, then $p(\mathcal{R}_1)$ also holds.

As shown below, the set of elements $\mathcal{R}$ can represent different objects related with $\mathcal{F}$, e.g. set of clauses or literals.

Definition 6. *(MSMP) The* Minimal Set over a Monotone Predicate *(MSMP) problem consists in finding a subset $\mathcal{M}$ of $\mathcal{R}$ such that $p(\mathcal{M})$ holds, and for any $\mathcal{M}' \subset \mathcal{M}$, $p(\mathcal{M}')$ does not hold, i.e. $\mathcal{M}$ is minimal.*

In [8,9], Bradley&Manna show that the problem of computing a prime implicate (from an existing implicate) can be represented as an instantiation of the MSMP problem. In addition, an algorithm for the MSMP problem is proposed, which is argued to be optimal (at least for the case when there exists one minimal set).

In this section we show that several other computational problems can be cast as instances of the MSMP problem.

Theorem 1. *Given a CNF formula $\mathcal{F}$, there exists an instantiation of the MSMP problem for each of the following problems:*

Table 1. Mappings to the MSMP Problem

	$\mathcal{R}$	$p(\mathcal{W})$, $\mathcal{W} \subseteq \mathcal{R}$
PI given c	$\{l \,\vert\, l \in c\}$	$\neg\mathsf{SAT}(\mathcal{F} \land \land_{l\in\mathcal{W}}\neg l)$
MM	$\mathsf{var}(\mathcal{F})$	$\mathsf{SAT}(\mathcal{F} \land \land_{x\in\mathsf{var}(\mathcal{F})\setminus\mathcal{W}}(\neg x))$
MUS	$\mathcal{F}$	$\neg\mathsf{SAT}(\mathcal{W})$
MCS	$\mathcal{F}$	$\mathsf{SAT}(\mathcal{F} \setminus \mathcal{W})$

1. *Computing a prime implicate (PI) of $\mathcal{F}$ given an implicate c of $\mathcal{F}$.*
2. *Computing a minimal model (MM) of $\mathcal{F}$.*
3. *Computing a minimal unsatisfiable subset (MUS) of $\mathcal{F}$.*
4. *Computing a minimal correction subset (MCS) of $\mathcal{F}$.*

Proof. Let $\mathcal{F}$ be a CNF formula. We consider each case separately.
1. For the case of a prime implicate of $\mathcal{F}$ given implicate c, the reference set is $\mathcal{R} = \{l \,\vert\, l \in c\}$. For $\mathcal{W} \subseteq \mathcal{R}$, define $p(\mathcal{W}) \triangleq \neg\mathsf{SAT}(\mathcal{F} \land \land_{l\in\mathcal{W}}\neg l)$. Clearly, $p(\mathcal{R})$ holds, because c is an implicate of $\mathcal{F}$. A minimal set $\mathcal{M} \subseteq \mathcal{R}$ such that $p(\mathcal{M})$ holds is a prime implicate of $\mathcal{F}$.
2. For the case of a minimal model, the reference set is $\mathcal{R} = \mathsf{var}(\mathcal{F})$. For $\mathcal{W} \subseteq \mathcal{R}$, define $p(\mathcal{W}) \triangleq \mathsf{SAT}(\mathcal{F} \land \land_{x\in\mathsf{var}(\mathcal{F})\setminus\mathcal{W}}(\neg x))$. Clearly, $p(\mathcal{R})$ holds if $\mathcal{F}$ is satisfiable, because no negated literals will be added to the formula. A minimal set $\mathcal{M} \subseteq \mathcal{R}$ such that $p(\mathcal{M})$ holds is a minimal model of $\mathcal{F}$. Observe that $\mathsf{var}(\mathcal{F}) \setminus \mathcal{M}$ represents a maximal set of literals that can be assigned value 0 while still satisfying the formula, and so the remaining literals represent a minimal model.
3. For the case of an MUS, the reference set is $\mathcal{R} = \mathcal{F}$ (or a subset known to be unsatisfiable). For $\mathcal{W} \subseteq \mathcal{R}$, define $p(\mathcal{W}) \triangleq \neg\mathsf{SAT}(\mathcal{W})$. Clearly, $p(\mathcal{R})$ holds if $\mathcal{F}$ is unsatisfiable. A minimal set $\mathcal{M} \subseteq \mathcal{R}$ such that $p(\mathcal{M})$ holds is an MUS of $\mathcal{F}$, since any subset will be satisfiable.
4. For the case of an MCS, the reference set is $\mathcal{R} = \mathcal{F}$. For $\mathcal{W} \subseteq \mathcal{R}$, define $p(\mathcal{W}) \triangleq \mathsf{SAT}(\mathcal{F}\setminus\mathcal{W})$. Clearly, $p(\mathcal{R})$ holds, because removing all clauses from a clause makes the (empty) formula satisfiable. A minimal set $\mathcal{M} \subseteq \mathcal{R}$ such that $p(\mathcal{M})$ holds is an MCS of $\mathcal{F}$, because then $\mathcal{F} \setminus \mathcal{M}$ is satisfiable, and for any $\mathcal{M}' \subset \mathcal{M}$, $\mathcal{F} \setminus \mathcal{M}'$ is unsatisfiable, particularly when $\mathcal{M}' = \mathcal{M} \setminus \{c\}$ for $c \in \mathcal{M}$.

To conclude the proof we note that the monotonicity of the predicate p in all cases above follows from the basic properties of CNF formulas and their models. □

Table 1 summarizes the mappings of the above computational problems to the MSMP problem. Theorem 1 can easily be extended to other computational problems. Concrete examples are minimal equivalent (or irredundant) subsets (MESes) and minimal distinguishing subsets (MDSes) [4]. Other immediate extensions are the problems studied above but for the case where clauses are handled as groups [21,26], e.g. group MUS, group MES, etc.

Algorithm 1. QUICKXPLAIN algorithm for the MSMP problem

Input: $\mathcal{B}$; $\mathcal{T}$, *has_set*
Output: Elements in the minimal set
if *has_set* $\wedge\, p(\mathcal{B})$ **then return** $\emptyset$
if $|\mathcal{T}| = 1$ **then return** $\mathcal{T}$
$m \leftarrow \lfloor \frac{|\mathcal{T}|}{2} \rfloor$
$(\mathcal{T}_1, \mathcal{T}_2) \leftarrow (\mathcal{T}_{1..m}, \mathcal{T}_{m+1..|\mathcal{T}|})$
$\mathcal{M}_2 \leftarrow$ QUICKXPLAIN$(\mathcal{B} \cup \mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1| > 0)$
$\mathcal{M}_1 \leftarrow$ QUICKXPLAIN$(\mathcal{B} \cup \mathcal{M}_2, \mathcal{T}_1, |\mathcal{M}_2| > 0)$
return $\mathcal{M}_1 \cup \mathcal{M}_2$

4.2 Algorithms for MSMP

Given the results of the previous section, and the algorithms proposed over the years for each of the above problems, one can conclude that many algorithms can be developed for the MSMP problem by adapting any existing algorithm to the framework of computing a minimal set over a monotone predicate. Clearly, some algorithms proposed for different problems correspond to the same algorithm in this framework. For example, the optimal algorithm proposed for Bradley&Manna for computing a prime implicate [8,9] corresponds to the well-known QUICKXPLAIN algorithm [19] for MUS extraction.

Given the wealth of algorithms proposed for MUS extraction [5,17], one can develop the following types of algorithms for the MSMP problem: (i) Insertion-based [13,32,22]; (ii) Deletion-based [11,3,22]; (iii) Dichotomic [18,17]; and (iv) QUICKXPLAIN [19]. Let m is the number of elements in the initial set of elements, and k is the size of the largest minimal set. In terms of predicate tests, insertion-based algorithms require a number of tests that ranges from $\mathcal{O}(m)$ [22] to $\mathcal{O}(m\,k)$ [13,32]. Deletion-based algorithms require $O(m)$ predicate tests. Dichotomic algorithms require $\mathcal{O}(k \log m)$ predicate tests. In the context of computing a prime implicate, [8,9] develop a QUICKXPLAIN-like algorithm. Algorithm 1 presents QUICKXPLAIN [19] adapted to the MSMP problem. $\mathcal{B}$ and $\mathcal{T}$ denote sets of elements, and the algorithm minimizes $\mathcal{T}$ with respect to a base $\mathcal{B}$. The initial values for $\mathcal{B}$ and $\mathcal{T}$ are, respectively, the $\emptyset$ and the original set of elements. The QUICKXPLAIN algorithm for MSMP iteratively splits the target set of elements ($\mathcal{T}$), and recursively calls itself. The predicate is tested when there exists another set of elements besides the current target. If the predicate is true, then the current target set is irrelevant and can be discarded. The base case corresponds to target sets of size 1, in which case the set is returned. As shown independently in [19] and in [8,9], the asymptotic number of predicate tests is $\mathcal{O}(k + k \log \frac{m}{k})$.

4.3 Pruning Predicate Tests

Recent work on MUS extraction is characterized by the development of several techniques to reduce the number of calls to a SAT oracle. Examples include

Algorithm 2. QUICKXPLAIN_CR MSMP with certificate refinement

Input: $\mathcal{B}$; $\mathcal{T}$, *has_set*
Output: $\mathcal{M}$; $\mathcal{C}$
if *has_set* **then**
  $(\mathsf{st}, \mathcal{C}) = p(\mathcal{B})$
  if *st* **then return** $(\emptyset, \mathcal{C})$
if $|\mathcal{T}| = 1$ **then return** $(\mathcal{T}, \mathcal{B})$
$m \leftarrow \lfloor \frac{|\mathcal{T}|}{2} \rfloor$
$(\mathcal{T}_1, \mathcal{T}_2) \leftarrow (\mathcal{T}_{1..m}, \mathcal{T}_{m+1..|\mathcal{T}|})$
$(\mathcal{M}_2, \mathcal{C}_1) \leftarrow$ QUICKXPLAIN_CR$(\mathcal{B} \cup \mathcal{T}_1, \mathcal{T}_2, |\mathcal{T}_1| > 0)$
$(\mathcal{M}_1, \mathcal{C}_2) \leftarrow$ QUICKXPLAIN_CR$(\mathcal{B} \cup \mathcal{M}_2, \mathcal{T}_1 \cap \mathcal{C}_1, |\mathcal{M}_2| > 0)$
return $(\mathcal{M}_1 \cup \mathcal{M}_2, (\mathcal{C}_1 \cup \mathcal{C}_2) \cap \mathcal{B})$

clause set refinement [14], redundancy removal [32] and model rotation [22,5,33]. Regarding these techniques, we state a few results for clause set refinement and model rotation, without proof; due to space restrictions this is beyond the scope of this paper. Clause set refinement for the MSMP problem can be used for PIs (given implicate c) and MUSes. For the case of MMs and MCSes, the equivalent notion is to keep only the clauses falsified by models. We refer to the generalized concept as *certificate refinement*. Certificate refinement requires the predicate test to return a certificate for the tested property being true. More precisely, $p(S)$ returns a pair $(\mathsf{st}, \mathcal{C})$ where st is true iff the tested property holds, and, $\mathcal{C} \subseteq S$ is such that that the property holds for $\mathcal{C}$ whenever st is true. If the property consists in CNF *unsatisfiability* (e.g. PIs and MUSes), then the certificate is an unsatisfiable core. In contrast, for CNF *satisfiability* (e.g. MMs and MCSes) the certificate is a set of falsified clauses.

Model rotation can be used for the PIs and MUSes. There is no equivalent concept for MMs and MCSes. The use of certificate refinement is illustrated for the case of QUICKXPLAIN in Algorithm 2. Although the algorithm may seem rather asymmetric in how it handles certificate refinement, recursion guarantees that certificates are used in most of the partitions made.

For the concrete cases of MUS extraction and PI computation, model rotation can also be integrated into the QUICKXPLAIN algorithm. Whenever the predicate does not hold (i.e. formula $\mathcal{B}$ is satisfiable), if the number of clauses in $\mathcal{T}$ is 1, or if the number of falsified clauses is 1, then a transition clause has been identified, and so model rotation can be applied. As shown in Section 6 for the case of MUSes, clause set refinement and model rotation serve to improve the basic algorithm. However, for the case of MUS extraction, QUICKXPLAIN performs significantly worse than most of the other algorithms considered.

5 Progression-Based Algorithms

This section describes a new algorithm for the MSMP problem, which is also specialized for the case of MUS extraction. Recent work on MUS extraction showed

that the most efficient algorithms are based on iteratively deciding with a SAT oracle whether each clause is in an MUS. This approach is referred to as deletion-based [11] MUS extraction and (more recently) as the hybrid approach [22]. In addition, a number of techniques are used to reduce the number of calls to a SAT oracle, namely clause set refinement [14], redundancy removal [32] and model rotation [22,5,33]. However, algorithms that in the worst case analyze all clauses are *not* optimal. As illustrated by the work of Junker [19] with the QuickXPlain algorithm for MUSes and Bradley&Manna [8,9] on PIs, algorithms can be developed that guarantee better worst-case asymptotic performance in terms of the number of SAT solver calls. Unfortunately, these algorithms perform poorly in practice. As shown by recent results [5,6], the algorithms with good theoretical properties tend to perform poorly when compared with recent high-performance algorithms [5]. (The experimental results in Section 6 confirm this observation.)

For MUS extraction, techniques such as clause set refinement allow dropping many clauses that are not included in an MUS. Similarly, model rotation often allows finding many clauses that must be in an MUS. Thus, in practice, the theoretical advantages of QuickXPlain are not observed. Moreover, in many settings, most of the clauses MUS algorithms end up analyzing are in the MUS. When the size of the MUS is close to the number of clauses that need to be analyzed, then QuickXPlain performs worse than approaches like the hybrid algorithm. Another drawback of an algorithm like QuickXPlain is that only a restricted version of model rotation can be integrated (see Section 4.3).

This section develops an algorithm for the MSMP problem that addresses all the drawbacks of the Bradley&Manna and QuickXPlain algorithms. The algorithm is shown to be correct, and the worst case number predicate tests is shown to be asymptotically equivalent to that of Bradley&Manna's and QuickXPlain algorithms. Afterwards, this section specializes the algorithm for the concrete case of MUS extraction, integrating techniques known to be essential for good performance [5].

5.1 A Progression Algorithm for the MSMP Problem

Algorithm 3 shows the organization of the new *progression*-based approach for the MSMP problem. The algorithm uses a geometric progression to define a subset of elements to drop from the set of elements on which the predicate is tested. If the predicate holds for the reduced set of elements, then the dropped elements are discarded, and the value of the progression is increased. Once the predicate does not hold, the algorithm invokes a binary search function (see Algorithm 4) to identify a *transition element* to include in the set of elements representing the minimal set. Similarly to the MUS case, a transition element is an element that, if dropped, the predicate does not hold. After identifying a transition element, the geometric progression is reset to 1, and the process continues. As will be shown after specializing Algorithm 3, standard pruning techniques are easily integrated in the progression-based algorithm. Some of the insights of the new algorithm include the following. First, dropping more than one element is achieved by the geometric progression. Second, identification of

Algorithm 3. Progression-based computation of a minimal set

Input: Working set $\mathcal{W}$
Output: Minimal set $\mathcal{M}$

$(\mathcal{M}, i) \leftarrow (\emptyset, 0)$
while $\mathcal{W} \neq \emptyset$ **do**
  $\nu \leftarrow \min(2^i, |\mathcal{W}|)$
  if $p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..\nu})$ **then**
    $\mathcal{W} \leftarrow \mathcal{W} \setminus \mathcal{W}_{1..\nu}$
    $i \leftarrow i + 1$
  else
    $j \leftarrow$ `BinSearchTransElem`$(\mathcal{M}, \mathcal{W}, \nu)$
    $\mathcal{W} \leftarrow \mathcal{W} \setminus \mathcal{W}_{1..j}$
    $\mathcal{M} \leftarrow \mathcal{M} \cup \mathcal{W}_{j..j}$
    $i \leftarrow 0$
return $\mathcal{M}$

Algorithm 4. Binary search for a transition element

function `BinSearchTransElem`$(\mathcal{M}, \mathcal{W}, \nu)$
  Input: $\mathcal{M}$; $\mathcal{W}$; ν
  Output: Index of transition element r
  $(l, r) \leftarrow (0, \nu)$
  while $l < r - 1$ **do** // *Inv:* $p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..l}) \wedge \neg p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..r})$
    $m \leftarrow \lfloor \frac{l+r}{2} \rfloor$
    if $p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..m})$ **then**
      $l \leftarrow m$
    else
      $r \leftarrow m$
  return r
end

the transition element is achieved using binary search. The progression is reset to 1, to simplify the complexity analysis; heuristics could be used to decide how to reset the progression after a transition element is identified.

5.2 Analysis of the Progression-Based Algorithm

This section analyzes Algorithm 3. The purpose is to show that the algorithm terminates, that it is correct (i.e. it computes a minimal set), and to prove the asymptotic complexity in terms of the number of times the predicate is tested. (Observe that, as illustrated in the previous section, the predicate is tested through a SAT solver call.) Before that, we need to analyze Algorithm 4.

Lemma 1. *Algorithm 4 terminates.*

Proof. If $l \geq r - 1$, then the while loop is not executed, and the algorithm terminates. If $l < r - 1$, then at each iteration of the while loop, either l or r are updated — the update would either increase l, or decrease r. □

Analysis of the pseudo-code allows concluding that the invariant $p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..l}) \wedge \neg p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..r})$ holds while executing *BinSearchTransElem*.

Lemma 2. *The value r returned by Algorithm 4 is the smallest index in the range $1..\nu$ such that $p(\mathcal{M} \cup (\mathcal{W} \setminus \mathcal{W}_{1..r-1}))$ holds and $p(\mathcal{M} \cup (\mathcal{W} \setminus \mathcal{W}_{1..r}))$ does not hold.*

Proposition 1. *Algorithm 3 terminates.*

Proof. By precondition (see Section 2), the sets considered are finite, and so is $\mathcal{W}$. At each execution of the while loop, either the predicate p holds or is does not. If the predicate holds, the set $\mathcal{W}$ is reduced by $\nu \geq 1$ elements. If the predicate does not hold, then the call to *BinSearchTransElem* (line 8) returns a value $1 \leq j \leq \nu$. Thus, set $\mathcal{W}$ is reduced by $j \geq 1$ elements. Therefore, the size of set $\mathcal{W}$ is reduced in each iteration of the while loop and so the algorithm terminates. □

Correctness requires conditions on set $\mathcal{W}$, besides being finite. Concretely, we require $p(\mathcal{W})$ to hold.

Proposition 2. *Algorithm 3 is correct, i.e. if $p(\mathcal{W})$ holds, then $p(\mathcal{M})$ holds and $\mathcal{M}$ is a minimal such subset of $\mathcal{W}$.*

Proof. Let $\mathcal{W}_o$ denote the original set given to the algorithm. We will show the invariant that $\mathcal{M}$ is a minimal subset of $\mathcal{W}_o \setminus \mathcal{W}$ s.t. $p(\mathcal{M} \cup \mathcal{W})$ holds. This invariant is sufficient to show the functional correctness since the algorithm terminates iff $\mathcal{W} = \emptyset$ and thus $\mathcal{M}$ is a minimal subset of $\mathcal{W}_o$ s.t. $p(\mathcal{M})$. The invariant holds upon initialization as $\mathcal{M} = \emptyset$ and thus $p(\mathcal{M} \cup \mathcal{W})$ holds by precondition and $\mathcal{M}$ is minimal. If $p(\mathcal{M} \cup \mathcal{W} \setminus \mathcal{W}_{1..\nu})$, then $\mathcal{W}_{1..\nu}$ is removed from $\mathcal{W}$ and thus at the end of the iteration $p(\mathcal{M} \cup \mathcal{W})$ still holds and $\mathcal{M}$ remains minimal since it has not been updated and at the same time $\mathcal{W}$ was shrunk. In the **else** branch of the **if** statement, j is computed s.t. $p(\mathcal{M} \cup (\mathcal{W} \setminus \mathcal{W}_{1..j-1}))$ holds but $p(\mathcal{M} \cup (\mathcal{W} \setminus \mathcal{W}_{1..j}))$ does *not* hold (by Lemma 2). Thus, removing $\mathcal{W}_{1..j}$ from $\mathcal{W}$ and adding $W_{j..j}$ to $\mathcal{M}$ preserves $p(\mathcal{M} \cup \mathcal{W})$. Moreover, it preserves the minimality of $\mathcal{M}$ since if $W_{j..j}$ were not inserted into $\mathcal{M}$, $p(\mathcal{M} \cup \mathcal{W})$ would not hold after $\mathcal{W}_{1..j}$ is removed from $\mathcal{W}$; and any other element cannot be removed from $\mathcal{M}$ since $\mathcal{M}$ was minimal by induction hypothesis, $W_{j..j}$ was part of the original W, and p is monotone. □

The complexity of the algorithm is measured in terms of the number of predicate tests (which correspond to calls to an NP, in our case SAT, oracle).

Proposition 3. *Let the size of $\mathcal{W}$ be $m = |\mathcal{W}|$ and the size of the largest minimal subset $\mathcal{M}$ be $k = |\mathcal{M}|$. If $k = 0$, Algorithm 3 requires $\mathcal{O}(\log(1 + m))$ tests of predicate p, and if $k > 0$, it requires $\mathcal{O}(k \log(1 + \frac{m}{k}))$ tests of predicate p.*

Proof. If $k = 0$, then the algorithm executes a simple geometric progression, and this gives the result. If $k > 0$, let $\mathcal{M} = \{\tau_1, \tau_2, \ldots, \tau_k\}$. Since Algorithm 3 analyzes the elements of $\mathcal{W}$ in order, the elements of $\mathcal{M}$ will also be discovered in order, first τ_1, then τ_2, and so on. For $i = 2, \ldots, k$, let α_i denote the number of elements of $\mathcal{W}$ between τ_{i-1} and τ_i, plus 1 due to τ_i. For $i = 1$, α_1 denotes the number of elements of $\mathcal{W}$ not in $\mathcal{M}$ before τ_1, plus 1 due to τ_1. We consider there are no elements above τ_k, since this gives the worst case (i.e. more elements located to the left of τ_k). Next consider the workings of Algorithm 3. While the predicate holds, the algorithm considers increasingly large subsets of elements of $\mathcal{W}$, starting with size 1 and progressing by powers of 2. For each i, the number of tests until the predicate does not hold (because the removed set contains τ_i) is $\log(1+\alpha_i)$, and the number of elements that need to be considered after τ_i is at most α_i. (We should use $\lceil \log(1 + \alpha_i) \rceil$, but this does not change the asymptotic result.) Binary search will then require $\mathcal{O}(\log(1 + \alpha_i))$ predicate tests to locate τ_i. (Again, we should use $\mathcal{O}(\lceil \log(1 + \alpha_i) \rceil)$, but this does not change the asymptotic result.) Thus, for each τ_i, the number of predicate tests is $\log(1 + \alpha_i)$, $i = 1, \ldots, k$. (Observe that for $k = i$ we are counting predicate tests that have been discarded, but this is correct in terms of computing an upper bound.) Summing over all i, we get $\sum_{i=1}^{k} \log(1 + \alpha_i)^2 \equiv 2 \log \Pi_{i=1}^{k}(1 + \alpha_i)$. Taking into consideration that $\sum_{i=1}^{k} \alpha_i = m$, the worst case corresponds to having sets of equal size, i.e. $\alpha_i = \frac{m}{k}$. Thus the worst case number of predicate tests is $2\,k \log(1 + \frac{m}{k}) = \mathcal{O}(k \log(1 + \frac{m}{k}))$. □

The asymptotic number of predicate tests can be shown to correspond to those of QuickXplain [19] and the optimal algorithm of Bradley&Manna [8,9], for both of which the number of predicate tests is $\mathcal{O}(k + k \log(\frac{m}{k}))$. If m is much larger than k, then the asymptotic number of tests is $O(k \log(\frac{m}{k}))$. If m is of the order of k, then the asymptotic number of tests is $O(k)$. Moreover, the progression-based algorithm incurs smaller constants for the case when $m = k$, i.e. when the reference set is a minimal set. In this situation, it is a factor of two better than either QuickXplain [19] or the optimal algorithm in [8,9].

5.3 Specialization for MUSes

This section shows how to specialize the progression-based MSMP algorithm to the case of MUS extraction. Observe that Algorithm 3 could be used, but it is possible to improve its performance in practice. The objective is to illustrate how two important pruning techniques can be integrated, namely clause-set refinement [14,22] and model rotation [22,5]. Other optimizations are also highlighted. Algorithm 5 shows the progression-based MUS extraction algorithm, whereas Algorithm 6 shows the binary search step. Several techniques to improve performance can be considered. Clause set refinement is applied each time the SAT solver call returns false. This is shown in line 6 in Algorithm 5 and lines 7 and 8 in Algorithm 6. Model rotation is applied each time a clause is added to the MUS set $\mathcal{M}$. This is shown in line 12 in Algorithm 5. Another improvement is to exploit each model computed by the SAT solver to reduce the

Algorithm 5. Progression-based computation of an MUS

Input: Unsatisfiable set $\mathcal{U} \subseteq \mathcal{F}$, viewed as sequence $\mathcal{U} = \langle c_1, \ldots, c_{|\mathcal{U}|} \rangle$
Output: MUS $\mathcal{M}$

1 $(\mathcal{M}, i) \leftarrow (\emptyset, 0)$
2 **while** $\mathcal{U} \neq \emptyset$ **do**
3 $\quad \nu \leftarrow \min(2^i, |\mathcal{U}|)$
4 $\quad (st, \mu, \mathcal{C}) \leftarrow \texttt{SAT}(\mathcal{M} \cup \mathcal{U} \setminus \mathcal{U}_{1..\nu})$
5 $\quad$ **if not** *st* **then**
6 $\quad\quad \mathcal{U} \leftarrow \mathcal{U} \cap \mathcal{C}$ `// Refine` $\mathcal{U}$
7 $\quad\quad i \leftarrow i + 1$
8 $\quad$ **else**
9 $\quad\quad \mathcal{T} \leftarrow \texttt{FalsifiedClauses}(\mu, \mathcal{U}_{1..\nu}\})$
10 $\quad\quad (c_j, \mu, \mathcal{U}) \leftarrow \texttt{BinSearchTransCl}(\mathcal{M}, \mathcal{U} \setminus \mathcal{T}, \mathcal{T}, \mu)$
11 $\quad\quad \mathcal{M} \leftarrow \mathcal{M} \cup \{c_j\}$
12 $\quad\quad \mathcal{U} \leftarrow \texttt{ModelRotate}(\mathcal{M}, \mathcal{U}, \mu)$
13 $\quad\quad i \leftarrow 0$
14 **return** $\mathcal{M}$

number of target clauses. For each computed model, the algorithms just need to subsequently search the transition clause over the clauses falsified by the model. This is achieved with function *FalsifiedClauses*. Observe that in Algorithm 6 the indices need to be corrected when the sets of clauses are changed. The additional functions ensure the correct indices are computed.

6 Experimental Results

This section presents the results of an experimental evaluation of a number of MUS extraction algorithms, including the progression-based algorithm presented in Section 5 [1]. Recent experimental results [5,6] have established the so-called *hybrid* algorithm, with clause set refinement and model rotation, to be the most efficient MUS extraction algorithm for practically-relevant benchmark sets. This section compares an implementation of the hybrid algorithm, with implementations of the progression-based algorithm (cf. Section 5) and the QuickXPlain algorithm [19] with various optimizations, as well as a number of additional state-of-the-art MUS extractors. The algorithms were evaluated on the benchmark instances from the MUS track of SAT Competition 2011 [2]. The experiments were performed on an HPC cluster, where each node is dual quad-core Intel Xeon E5450 3 GHz with 32 GB of RAM. Each algorithm was run with a timeout of 3600 seconds and a memory limit of 4 GB per input instance.

Figure 1 presents a cactus plot comparing the performance of the following MUS extractors: (*i*) our implementation of the QuickXPlain algorithm (Algorithm 1,

[1] Given the results in this paper, we could have also considered PIs, MMs, MCSes, MESes, etc. but opted to focus on MUS extraction.
[2] `http://www.satcompetition.org/`

Algorithm 6. Binary search for transition clause

```
function BinSearchTransCl(M, U, T, μ)
    Input: M; U; T ⊆ U; μ
    Output: Index of transition clause r; μ_R; U
1   μ_R ← μ
2   (l, r) ← (0, |T|)
3   while l < r − 1 do
4       m ← ⌊(l+r)/2⌋
5       (st, μ, C) ← SAT(M ∪ U ∪ T \ T_{1..m})
6       if not st then
7           U ← U ∩ C                                   // Refine U
8           (l, r, T) ← DropNonCoreClauses(C, T, l, r)
9       else
10          (l, r, T) ← FalsifiedClauses(μ, T, l, r)
11          μ_R ← μ          // Save model for model rotation later
12  return (c_r, μ_R, T ∪ U)
end
```

denoted `QuickXPlain` in the plot), additionally with the certificate refinement for MUSes (Algorithm 2, denoted `QuickXPlain+UC`), and also with specialized version of model rotation (denoted `QuickXPlain+UC+ROT`); (*ii*) the top three MUS extractors from SAT Competition 2011, namely `MoUsSaka` [20] and `Haifa-MUC` [29] with and without preprocessing; (*iii*) the implementation of the hybrid and the dichotomic algorithms in the state-of-the-art MUS extractor `MUSer2` [6], denoted as `HYB` and `DICH`, respectively; (*iv*) the implementation of the progression-based algorithm (Algorithm 3, denoted `PROG`), also in `MUSer2`.

A number of conclusions can be drawn from the plot in Figure 1. First we note that the progression-based algorithm outperforms the hybrid algorithm, which, to our knowledge, is the best performing MUS extraction algorithm to date [5,6]. This latter claim is additionally supported by the fact that the top three MUS extractors from the SAT Competition 2011 trail significantly behind both `HYB` and `PROG` in Figure 1. This result is significant, since it implies that the progression-based algorithm is the first algorithm that is both asymptotically optimal, and that also performs well in practice. The experimental results are also very clear about the significant performance difference between the new algorithm and QuickXplain, even with the addition of the certificate refinement and model rotation. As a side note, we point out that these optimizations, which have not been proposed in previous work, have a notable positive impact on the performance of QuickXplain.

The scatter plots in Figure 2 provide additional insights into performance profile of the progression-based algorithm. Comparing the algorithm with QuickXplain (left plot), we conclude that the new algorithm is a clear win, even when QuickXplain is augmented with the proposed optimizations. This suggests that the new algorithm captures the optimal behavior of the recursive partitioning

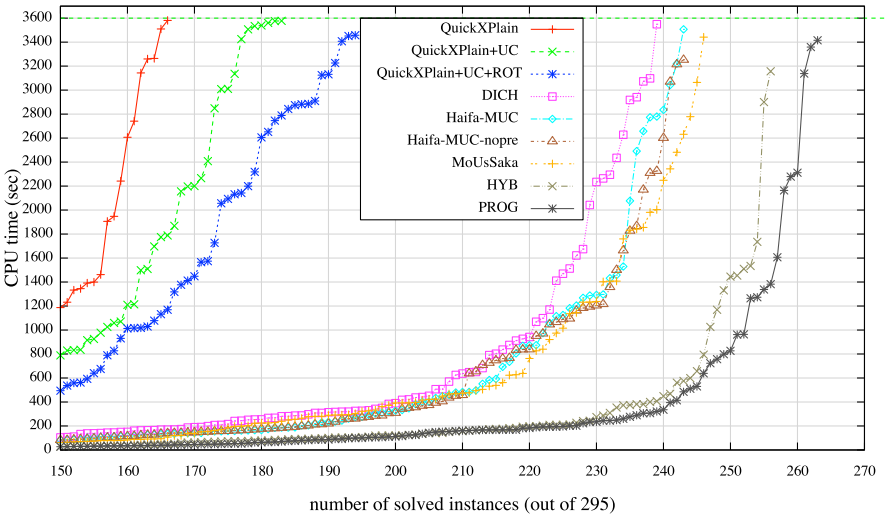


Fig. 1. Cactus plot: CPU runtimes of selected extractors on the benchmarks from the MUS track of SAT Competition 2011. Time limit 3600 sec, memory limit 4 GB.

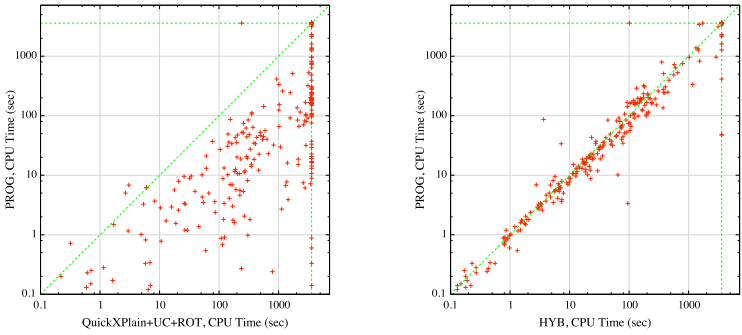


Fig. 2. CPU runtime of MUS extraction. Left: progression-based vs QuickXPlain with optimizations. Right: progression-based algorithm vs hybrid.

scheme employed by QuickXPlain, while at the same time avoiding the unfavorable for QuickXPlain cases of instances with few non-MUS clauses. The comparison with the hybrid algorithm (right plot) confirms the overall positive trend towards the progression-based algorithm, but also shows that the performance of the two algorithms might be complementary, suggesting a possible integration of the algorithms into a portfolio.

7 Conclusions and Research Directions

This paper shows that several computational problems on Boolean formulas can be formulated as computing a minimal set subject to a monotone predicate,

i.e. the MSMP problem. Examples include prime implicates, minimal models, minimal unsatisfiable subsets, minimal correction subsets, among others. This result allows using the same algorithms to solve all of these problems. In addition, the paper summarizes how standard pruning techniques can be adapted to each concrete computational problem. The insights provided by this result motivate the development of a new optimal algorithm for the MSMP problem, which is asymptotically as efficient as the asymptotically best algorithms. More importantly, the new algorithm has smaller constants than other algorithms, and is amenable to the integration of well-known pruning techniques. The paper also shows how the new algorithm for the MSMP problem can be specialized for the case of MUS extraction. Experimental results, obtained on representative MUS problem instances, demonstrate that the new progression-based algorithm outperforms another optimal algorithm, namely an optimized version of QuickXPlain [19], that includes clause set refinement and model rotation. The experimental results also demonstrate that the new algorithm outperforms the current state of the art hybrid algorithm [5,6].

The results in this paper open several research directions. For example, how does the new algorithm for the MSMP problem perform on other problems, e.g. prime implicates, minimal models, minimal correction subsets among others? How to apply the results in this paper to the case of group MUS, etc.?

References

1. Andraus, Z.S., Liffiton, M.H., Sakallah, K.A.: Reveal: A formal verification tool for verilog designs. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 343–352. Springer, Heidelberg (2008)
2. Bailey, J., Stuckey, P.J.: Discovery of minimal unsatisfiable subsets of constraints using hitting set dualization. In: Hermenegildo, M.V., Cabeza, D. (eds.) PADL 2004. LNCS, vol. 3350, pp. 174–186. Springer, Heidelberg (2005)
3. Bakker, R.R., Dikker, F., Tempelman, F., Wognum, P.M.: Diagnosing and solving over-determined constraint satisfaction problems. In: IJCAI, pp. 276–281 (1993)
4. Belov, A., Janota, M., Lynce, I., Marques-Silva, J.: On computing minimal equivalent subformulas. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 158–174. Springer, Heidelberg (2012)
5. Belov, A., Lynce, I., Marques-Silva, J.: Towards efficient MUS extraction. AI Commun. 25(2), 97–116 (2012)
6. Belov, A., Marques-Silva, J.: MUSer2: An efficient MUS extractor, system description. JSAT 8, 123–128 (2012)
7. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. IOS Press (2009)
8. Bradley, A.R., Manna, Z.: Checking safety by inductive generalization of counterexamples to induction. In: FMCAD, pp. 173–180 (2007)
9. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Asp. Comput. 20(4-5), 379–405 (2008)
10. Cadoli, M., Donini, F.M.: A survey on knowledge compilation. AI Commun. 10(3-4), 137–150 (1997)
11. Chinneck, J.W., Dravnieks, E.W.: Locating minimal infeasible constraint sets in linear programs. INFORMS Journal on Computing 3(2), 157–168 (1991)

12. Darwiche, A., Marquis, P.: A knowledge compilation map. J. Artif. Intell. Res. (JAIR) 17, 229–264 (2002)
13. de Siqueira, J.L.,, N., Puget, J.-F.: Explanation-based generalisation of failures. In: ECAI, pp. 339–344 (1988)
14. Dershowitz, N., Hanna, Z., Nadel, A.: A scalable algorithm for minimal unsatisfiable core extraction. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 36–41. Springer, Heidelberg (2006)
15. Felfernig, A., Schubert, M., Zehentner, C.: An efficient diagnosis algorithm for inconsistent constraint sets. AI EDAM 26(1), 53–62 (2012)
16. Gomes, C.P., Kautz, H., Sabharwal, A., Selman, B.: Satisfiability solvers. In: Frank van Harmelen, V.L., Porter, B. (eds.) Handbook of Knowledge Representation, vol. 3, pp. 89–134 (2008)
17. Grégoire, É., Mazure, B., Piette, C.: On approaches to explaining infeasibility of sets of Boolean clauses. In: ICTAI, pp. 74–83 (November 2008)
18. Hemery, F., Lecoutre, C., Sais, L., Boussemart, F.: Extracting MUCs from constraint networks. In: ECAI, pp. 113–117 (2006)
19. Junker, U.: QUICKXPLAIN: Preferred explanations and relaxations for over-constrained problems. In: AAAI, pp. 167–172 (2004)
20. Kottler, S.: Description of the SApperloT, SArTagnan and MoUsSaka solvers for the SAT-Competition 2011 (2011), `http://www.satcompetition.org/2011/`
21. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reasoning 40(1), 1–33 (2008)
22. Marques-Silva, J., Lynce, I.: On improving MUS extraction algorithms. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 159–173. Springer, Heidelberg (2011)
23. Marques-Silva, J., Lynce, I., Malik, S.: Conflict-driven clause learning SAT solvers. In: Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.) Handbook of Satisfiability, pp. 131–153. IOS Press (2009)
24. Marquis, P.: Consequence finding algorithms. In: Algorithms for Defeasible and Uncertain Reasoning. Kluwer Academic Publishers (2000)
25. McCarthy, J.: Circumscription - a form of non-monotonic reasoning. Artif. Intell. 13(1-2), 27–39 (1980)
26. Nadel, A.: Boosting minimal unsatisfiable core extraction. In: FMCAD, pp. 121–128 (October 2010)
27. Nöhrer, A., Biere, A., Egyed, A.: Managing SAT inconsistencies with HUMUS. In: VaMoS, pp. 83–91 (2012)
28. Reiter, R.: A theory of diagnosis from first principles. Artif. Intell. 32(1), 57–95 (1987)
29. Ryvchin, V., Strichman, O.: Faster extraction of high-level minimal unsatisfiable cores. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 174–187. Springer, Heidelberg (2011)
30. Schlobach, S., Huang, Z., Cornet, R., van Harmelen, F.: Debugging incoherent terminologies. J. Autom. Reasoning 39(3), 317–349 (2007)
31. Soh, T., Inoue, K.: Identifying necessary reactions in metabolic pathways by minimal model generation. In: ECAI, pp. 277–282 (2010)
32. van Maaren, H., Wieringa, S.: Finding guaranteed mUSes fast. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 291–304. Springer, Heidelberg (2008)
33. Wieringa, S.: Understanding, improving and parallelizing MUS finding using model rotation. In: Milano, M. (ed.) CP 2012. LNCS, vol. 7514, pp. 672–687. Springer, Heidelberg (2012)

A Scalable and Nearly Uniform Generator of SAT Witnesses*

Supratik Chakraborty[1], Kuldeep S. Meel[2], and Moshe Y. Vardi[2]

[1] Indian Institute of Technology Bombay, India
[2] Department of Computer Science, Rice University

Abstract. Functional verification constitutes one of the most challenging tasks in the development of modern hardware systems, and simulation-based verification techniques dominate the functional verification landscape. A dominant paradigm in simulation-based verification is directed random testing, where a model of the system is simulated with a set of random test stimuli that are uniformly or near-uniformly distributed over the space of all stimuli satisfying a given set of constraints. Uniform or near-uniform generation of solutions for large constraint sets is therefore a problem of theoretical and practical interest. For Boolean constraints, prior work offered heuristic approaches with no guarantee of performance, and theoretical approaches with proven guarantees, but poor performance in practice. We offer here a new approach with theoretical performance guarantees and demonstrate its practical utility on large constraint sets.

1 Introduction

Functional verification constitutes one of the most challenging tasks in the development of modern hardware systems. Despite significant advances in formal verification over the last few decades, there is a huge mismatch between the sizes of industrial systems and the capabilities of state-of-the-art formal verification tools [6]. Simulation-based verification techniques therefore dominate the functional verification landscape [8]. A dominant paradigm in simulation-based verification is directed random testing. In this paradigm, an operational (usually, low-level) model of the system is simulated with a set of random test stimuli satisfying a set of *constraints* [7,18,23]. The simulated behavior is then compared with the expected behavior, and any mismatch is flagged as indicative of a bug. The constraints that stimuli must satisfy typically arise from various sources such as domain and application-specific knowledge, architectural and environmental

* Work supported in part by NSF grants CNS 1049862 and CCF-1139011, by NSF Expeditions in Computing project "ExCAPE: Expeditions in Computer Augmented Program Engineering," by BSF grant 9800096, by a gift from Intel, by a grant from Board of Research in Nuclear Sciences, India, and by the Shared University Grid at Rice funded by NSF under Grant EIA-0216467, and a partnership between Rice University, Sun Microsystems, and Sigma Solutions, Inc.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 608–623, 2013.

requirements, specifications of corner-case scenarios, and the like. Test requirements from these varied sources are compiled into a set of constraints and fed to a constraint solver to obtain test stimuli. Developing constraint solvers (and test generators) that can reason about large sets of constraints is therefore an extremely important activity for industrial test and verification applications [13].

Despite the diligence and insights that go into developing constraint sets for generating directed random tests, the complexity of modern hardware systems makes it hard to predict the effectiveness of any specific test stimulus. It is therefore common practice to generate a large number of stimuli satisfying a set of constraints. Since every stimulus is *a priori* as likely to expose a bug as any other stimulus, it is desirable to sample the solution space of the constraints uniformly or near-uniformly (defined formally below) at random [18]. A naive way to accomplish this is to first generate all possible solutions, and then sample them uniformly. Unfortunately, generating all solutions is computationally prohibitive (and often infeasible) in practical settings of directed random testing. For example, we have encountered systems of constraints where the expected number of solutions is of the order of 2^{100}, and there is no simple way of deriving one solution from another. It is therefore interesting to ask: *Given a set of constraints, can we sample the solution space uniformly or near-uniformly, while scaling to problem sizes typical of testing/verification scenarios?* An affirmative answer to this question has implications not only for directed random testing, but also for other applications like probabilistic reasoning, approximate model counting and Markov logic networks [4,19].

In this paper, we consider Boolean constraints in conjunctive normal form (CNF), and address the problem of near-uniform generation of their solutions, henceforth called *SAT Witnesses*. This problem has been of long-standing theoretical interest [20,21]. Industrial approaches to solving this problem either rely on ROBDD-based techniques [23], which do not scale well (see, for example, the comparison in [16]), or use heuristics that offer no guarantee of performance or uniformity when applied to large problem instances[1]. Prior published work in this area broadly belong to one of two categories. In the first category [22,15,12,16], the focus is on heuristic sampling techniques that scale to large systems of constraints. Monte Carlo Markov Chain (MCMC) methods and techniques based on random seedings of SAT solvers belong to this category. However, these methods either offer very weak or no guarantees on the uniformity of sampling (see [16] for a comparison), or require the user to provide hard-to-estimate problem-specific parameters that crucially affect the performance and uniformity of sampling. In the second category of work [5,14,23], the focus is on stronger guarantees of uniformity of sampling. Unfortunately, our experience indicates that these techniques do not scale even to relatively small problem instances (involving few tens of variables) in practice.

The work presented in this paper tries to bridge the above mentioned extremes. Specifically, we provide guarantees of near-uniform sampling, and of a bounded probability of failure, without the user having to provide any

[1] Private communication: R. Kurshan

hard-to-estimate parameters. We also demonstrate that our algorithm scales in practice to constraints involving thousands of variables. Note that there is evidence that uniform generation of SAT witnesses is harder than SAT solving [14]. Thus, while today's SAT solvers are able to handle hundreds of thousands of variables and more, we believe that scaling of our algorithm to thousands of variables is a major improvement in this area. Since a significant body of constraints that arise in verification settings and in other application areas (like probabilistic reasoning) can be encoded as Boolean constraints, our work opens new directions in directed random testing and in these application areas.

The remainder of the paper is organized as follows. In Section 2, we review preliminaries and notation needed for the subsequent discussion. In Section 3, we give an overview of some algorithms presented in earlier work that come close to our work. Design choices behind our algorithm, some implementation issues, and a mathematical analysis of the guarantees provided by our algorithm are discussed in Section 4. Section 5 discusses experimental results on a large set of benchmarks. Our experiments demonstrate that our algorithm is more efficient in practice and generates witnesses that are more evenly distributed than those generated by the best known alternative algorithm that scales to comparable problem sizes. Finally, we conclude in Section 6.

2 Notation and Preliminaries

Our algorithm can be viewed as an adaptation of the algorithm proposed by Bellare, Goldreich and Petrank [5] for uniform generation of witnesses for $\mathcal{NP}$-relations. In the remainder of the paper, we refer to Bellare et al.'s algorithm as the BGP algorithm (after the last names of the authors). Our algorithm also has similarities with algorithms presented by Gomes, Sabharwal and Selman [12] for near-uniform sampling of SAT witnesses. We begin with some notation and preliminaries needed to understand these related work.

Let Σ be an alphabet and $R \subseteq \Sigma^* \times \Sigma^*$ be a binary relation. We say that R is an $\mathcal{NP}$-relation if R is polynomial-time decidable, and if there exists a polynomial $p(\cdot)$ such that for every $(x, y) \in R$, we have $|y| \leq p(|x|)$. Let L_R be the language $\{x \in \Sigma^* \mid \exists y \in \Sigma^*, (x, y) \in R\}$. The language L_R is said to be in $\mathcal{NP}$ if R is an $\mathcal{NP}$-relation. The set of all satisfiable propositional logic formulae in CNF is known to be a language in $\mathcal{NP}$. Given $x \in L_R$, a *witness* of x is a string $y \in \Sigma^*$ such that $(x, y) \in R$. The set of all witnesses of x is denoted R_x. For notational convenience, let us fix Σ to be $\{0, 1\}$ without loss of generality. If R is an $\mathcal{NP}$-relation, we may further assume that for every $x \in L_R$, every witness $y \in R_x$ is in $\{0, 1\}^n$, where $n = p(|x|)$ for some polynomial $p(\cdot)$.

Given an $\mathcal{NP}$ relation R, a *probabilistic generator* of witnesses for R is a probabilistic algorithm $\mathcal{G}(\cdot)$ that takes as input a string $x \in L_R$ and generates a random witness of x. Throughout this paper, we use $\mathsf{Pr}[X]$ to denote the probability of outcome X of sampling from a probability space. A *uniform generator* $\mathcal{G}^u(\cdot)$ is a probabilistic generator that guarantees $\mathsf{Pr}[\mathcal{G}^u(x) = y] = 1/|R_x|$ for every witness y of x. A *near-uniform generator* $\mathcal{G}^{nu}(\cdot)$ relaxes the guarantee of

uniformity, and ensures that $\mathsf{Pr}\left[\mathcal{G}^{nu}(x) = y\right] \geq c \cdot (1/|R_x|)$ for a constant c, where $0 < c \leq 1$. Clearly, the larger c is, the closer a near-uniform generator is to being a uniform generator. Note that near-uniformity, as defined above, is a more relaxed approximation of uniformity compared to the notion of "almost uniformity" introduced in [5,14]. In the present work, we sacrifice the guarantee of uniformity and settle for a near-uniform generator in order to gain performance benefits. Our experiments, however, show that the witnesses generated by our algorithm are fairly uniform in practice. Like previous work [5,14], we allow our generator to occasionally "fail", i.e. the generator may occasionaly output no witness, but a special failure symbol $\perp$. A generator that occasionally fails must have its failure probability bounded above by d, where d is a constant strictly less than 1.

A key idea in the BGP algorithm for uniform generation of witnesses for $\mathcal{NP}$-relations is to use r-wise independent hash functions that map strings in $\{0,1\}^n$ to $\{0,1\}^m$, for $m \leq n$. The objective of using these hash functions is to partition R_x with high probability into a set of "well-balanced" and "small" cells. We follow a similar idea in our work, although there are important differences. Borrowing related notation and terminology from [5], we give below a brief overview of r-wise independent hash functions as used in our context.

Let n, m and r be positive integers, and let $H(n,m,r)$ denote a family of r-wise independent hash functions mapping $\{0,1\}^n$ to $\{0,1\}^m$. We use $h \xleftarrow{R} H(n,m,r)$ to denote the act of choosing a hash function h uniformly at random from $H(n,m,r)$. By virtue of r-wise independence, for each $\alpha_1, \ldots \alpha_r \in \{0,1\}^m$ and for each distinct $y_1, \ldots y_r \in \{0,1\}^n$, $\mathsf{Pr}\left[\bigwedge_{i=1}^{r} h(y_i) = \alpha_i : h \xleftarrow{R} H(n,m,r)\right] = 2^{-mr}$.

For every $\alpha \in \{0,1\}^m$ and $h \in H(n,m,r)$, let $h^{-1}(\alpha)$ denote the set $\{y \in \{0,1\}^n \mid h(y) = \alpha\}$. Given $R_x \subseteq \{0,1\}^n$ and $h \in H(n,m,r)$, we use $R_{x,h,\alpha}$ to denote the set $R_x \cap h^{-1}(\alpha)$. If we keep h fixed and let α range over $\{0,1\}^m$, the sets $R_{x,h,\alpha}$ form a partition of R_x. Following the notation of Bellare et al., we call each element of such a parition a *cell* of R_x induced by h. It has been argued in [5] that if h is chosen uniformly at random from $H(n,m,r)$ for $r \geq 1$, the expected size of $R_{x,h,\alpha}$, denoted $\mathsf{E}\left[|R_{x,h,\alpha}|\right]$, is $|R_x|/2^m$, for each $\alpha \in \{0,1\}^m$.

In [5], the authors suggest using polynomials over finite fields to generate r-wise independent hash functions. We call these *algebraic* hash functions. Choosing a random algebraic hash function $h \in H(n,m,r)$ requires choosing a sequence $(a_0, \ldots a_{r-1})$ of elements in the field $\mathbb{F} = \mathrm{GF}(2^{\max(n,m)})$, where $GF(2^k)$ denotes the Galois field of 2^k elements. Given $y \in \{0,1\}^n$, the hash value $h(y)$ can be computed by interpretting y as an element of $\mathbb{F}$, computing $\Sigma_{j=0}^{r-1} a_j y^j$ in $\mathbb{F}$, and selecting m bits of the encoding of the result. The authors of [5] suggest polynomial-time optimizations for operations in the field $\mathbb{F}$. Unfortunately, even with these optimizations, computing algebraic hash functions is quite expensive in practice when non-linear terms are involved, as in $\Sigma_{j=0}^{r-1} a_j y^j$,

Our approach uses computationally efficient linear hash functions. As we show later, pairwise independent hash functions suffice for our purposes. The literature describes several families of efficiently computable pairwise independent hash

functions. One such family, which we denote $H_{conv}(n, m, 2)$, is based on the *wrapped convolution* function [17]. For $a \in \{0,1\}^{n+m-1}$ and $y \in \{0,1\}^n$, the wrapped convolution $c = (a \bullet y)$ is defined as an element of $\{0,1\}^m$ as follows: for each $i \in \{1, \ldots m\}$, $c[i] = \bigoplus_{j=1}^{n}(y[j] \wedge a[i+j-1])$, where $\bigoplus$ denotes logical xor and $v[i]$ denotes the i^{th} component of the bit-vector v. The family $H_{conv}(n, m, 2)$ is defined as $\{h_{a,b}(y) = (a \bullet y) \oplus_m b \mid a \in \{0,1\}^{n+m-1}, b \in \{0,1\}^m\}$, where $\oplus_m$ denotes componentwise xor of two elements of $\{0,1\}^m$. By randomly choosing a and b, we can randomly choose a function $h_{a,b}(x)$ from this family. It has been shown in [17] that $H_{conv}(n, m, 2)$ is pairwise independent. Our implementation of a near-uniform generator of CNF SAT witnesses uses $H_{conv}(n, m, 2)$.

3 Related Algorithms in Prior Work

We now discuss two algorithms that are closely related to our work. In 1998, Bellare et al. [5] proposed the BGP algorithm, showing that uniform generation of $\mathcal{NP}$-witnesses can be achieved in probabilistic polynomial time using an $\mathcal{NP}$-oracle. This improved on previous work by Jerrum, Valiant and Vazirani [14], who showed that uniform generation can be achieved in probabilistic polynomial time using a Σ_2^P oracle, and almost-uniform generation (as defined in [14]) can be achieved in probabilistic polytime using an $\mathcal{NP}$ oracle.

Let R be an $\mathcal{NP}$-relation over Σ. The BGP algorithm takes as input an $x \in L_R$ and either generates a witness that is uniformly distributed in R_x, or produces a symbol $\perp$ (indicating a failed run). The pseudocode for the algorithm is presented below. In the presentation, we assume w.l.o.g. that n is an integer such that $R_x \subseteq \{0,1\}^n$. We also assume access to $\mathcal{NP}$-oracles to answer queries about cardinalities of witness sets and also to enumerate small witness sets.

Algorithm BGP(x) :
/* Assume $R_x \subseteq \{0,1\}^n$ */
1: pivot $\leftarrow 2n^2$;
2: **if** ($|R_x| \leq$ pivot)
3: List all elements $y_1, \ldots y_{|R_x|}$ of R_x;
4: Choose j at random from $\{1, \ldots |R_x|\}$, and **return** y_j;
5: **else**
6: $l \leftarrow 2\lceil \log_2 n \rceil$; $i \leftarrow l - 1$;
7: **repeat**
8: $i \leftarrow i + 1$;
9: Choose h at random from $H(n, i - l, n)$;
10: **until** $(\forall \alpha \in \{0,1\}^{i-l}, |R_{x,h,\alpha}| \leq 2n^2)$ or $(i = n - 1)$;
11: **if** $(\exists \alpha \in \{0,1\}^{i-l}, |R_{x,h,\alpha}| > 2n^2)$ **return** $\perp$;
12: Choose α at random from $\{0,1\}^{i-l}$;
13: List all elements $y_1, \ldots y_{|R_{x,h,\alpha}|}$ of $R_{x,h,\alpha}$;
14: Choose j at random from $\{1, \ldots \text{pivot}\}$;
15: **if** $j \leq |R_{x,h,\alpha}|$, **return** y_j;
16: **else return** $\perp$;

For clarity of exposition, we have made a small adaptation to the algorithm originally presented in [5]. Specifically, if h does not satisfy $(\forall \alpha \in \{0,1\}^{i-l}, |R_{x,h,\alpha}| \leq 2n^2)$ when the loop in lines 7–10 terminates, the original algorithm forces a specific choice of h. Instead, algorithm BGP simply outputs $\perp$ (indicating a failed run) in this situation. A closer look at the analysis presented in [5] shows that all results continue to hold with this adaptation. The authors of [5] use algebraic hash functions and random choices of n-tuples in $\mathrm{GF}(2^n)$ to implement the selection of a random hash function in line 9 of the pseudocode. The following theorem summarizes the key properties of the BGP algorithm [5].

Theorem 1. *If a run of the* BGP *algorithm is successful, the probability that $y \in R_x$ is generated by the algorithm is independent of y. Further, the probability that a run of the algorithm fails is ≤ 0.8.*

Since the probability of any witness $y \in R_x$ being output by a successful run of the algorithm is independent of y, the BGP algorithm guarantees uniform generation of witnesses. However, as we argue in the next section, scaling the algorithm to even medium-sized problem instances is quite difficult in practice. Indeed, we have found no published report discussing any implementation of the BGP algorithm.

In 2007, Gomes et al. [12] presented two closely related algorithms named XORSample and XORSample′ for near-uniform sampling of combinatorial spaces. A key idea in both these algorithms is to constrain a given instance F of the CNF SAT problem by a set of randomly selected xor constraints over the variables appearing in F. An xor constraint over a set V of variables is an equation of the form $e = c$, where $c \in \{0,1\}$ and e is the logical xor of a subset of V. A probability distribution $\mathbb{X}(|V|, q)$ over the set of all xor constraints over V is characterized by the probability q of choosing a variable in V. A random xor constraint from $\mathbb{X}(|V|, q)$ is obtained by forming an xor constraint where each variable in V is chosen independently with probability q, and c is chosen uniformly at random.

We present the pseudocode of algorithm XORSample′ below. The algorithm uses a function SATModelCount that takes a Boolean formula F and returns the exact count of witnesses of F. Algorithm XORSample′ takes as inputs a CNF formula F, the parameter q discussed above and an integer $s > 0$. Suppose the number of variables in F is n. The algorithm proceeds by conjoining s xor constraints to F, where the constraints are chosen randomly from the distribution $\mathbb{X}(n, q)$. Let F' denote the conjunction of F and the random xor constraints, and let mc denote the model count (i.e., number of witnesses) of F'. If $mc \geq 1$, the algorithm enumerates the witnesses of F' and chooses one witness at random. Otherwise, the algorithm outputs $\perp$, indicating a failed run.

Algorithm XORSample′(F, q, s)
/* n = Number of variables in F */
1: $Q_s \leftarrow \{s$ random xor constraints from $\mathbb{X}(n, q)\}$;
2: $F' = F \wedge (\bigwedge_{f \in Q_s} f)$;
3: $mc \leftarrow$ SATModelCount(F');
4: **if** $(mc \geq 1)$

5: Choose i at random from $\{1, \ldots mc\}$;
6: List the first i witnesses of F';
7: **return** i^{th} witness of F';
8: **else return** $\perp$;

Algorithm XORSample can be viewed as a variant of algorithm XORSample′ in which we check if mc is exactly 1 (instead of $mc \geq 1$) in line 4 of the pseudocode. An additional difference is that if the check in line 4 fails, algorithm XORSample starts afresh from line 1 by randomly choosing s xor constraints. In our experiments, we observed that XORSample′ significantly outperforms XORSample, hence we consider only XORSample′ for comparison with our algorithm. The following theorem is proved in [12]

Theorem 2. *Let F be a Boolean formula with 2^{s^*} solutions. Let α be such that $0 < \alpha < s^*$ and $s = s^* - \alpha$. For a witness y of F, the probability with which* XORSample′ *with parameters $q = \frac{1}{2}$ and s outputs y is bounded below by $c'(\alpha)2^{-s^*}$, where $c'(\alpha) = \frac{1-2^{-\alpha/3}}{(1+2^{-\alpha})(1+2^{-\alpha/3})}$. Further,* XORSample′ *succeeds with probability larger than $c'(\alpha)$.*

While the choice of $q = \frac{1}{2}$ allowed the authors of [12] to prove Theorem 2, the authors acknowledge that finding witnesses of F' is quite hard in practice when random xor constraints are chosen from $\mathbb{X}(n, \frac{1}{2})$. Therefore, they advocate using values of q much smaller than $\frac{1}{2}$. Unfortunately, the analysis that yields the theoretical guarantees in Theorem 2 does not hold with these smaller values of q. This illustrates the conflict between witness generators with good performance in practice, and those with good theoretical guarantees.

4 The UniWit Algorithm: Design and Analysis

We now describe an adaptation, called UniWit, of the BGP algorithm that scales to much larger problem sizes than those that can be handled by the BGP algorithm, while weakening the guarantee of uniform generation to that of near-uniform generation. Experimental results indicate that the witnesses generated by our algorithm are fairly uniform in practice. Our algorithm can also be viewed as an adaptation of the XORSample′ algorithm, in which we do not need to provide hard-to-estimate problem-specific parameters like s and q.

We begin with some observations about the BGP algorithm. In what follows, line numbers refer to those in the pseudocode of the BGP algorithm presented in Section 3. Our first observation is that the loop in lines 7–10 of the pseudocode iterates until either $|R_{x,h,\alpha}| \leq 2n^2$ for *every* $\alpha \in \{0,1\}^{i-l}$ or i increments to $n-1$. Checking the first condition is computationally prohibitive even for values of $i-l$ and n as small as a few tens. So we ask if this condition can be simplified, perhaps with some weakening of theoretical guarantees. Indeed, we have found that if the condition requires that $1 \leq |R_{x,h,\alpha}| \leq 2n^2$ for a *specific* $\alpha \in \{0,1\}^{i-l}$ (instead of for *every* $\alpha \in \{0,1\}^{i-l}$), we can still guarantee near-uniformity (but not uniformity) of the generated witnesses. This suggests choosing both a random $h \in H(n, i-l, n)$ *and* a random $\alpha \in \{0,1\}^{i-l}$ *within* the loop of lines 7–10.

The analysis presented in [5] relies on h being sampled uniformly from a family of n-wise independent hash functions. In the context of generating SAT witnesses, n denotes the number of propositional variables in the input formula. This can be large (several thousands) in problems arising from directed random testing. Unfortunately, implementing n-wise independent hash functions using algebraic hash functions (as advocated in [5]) for large values of n is computationally infeasible in practice. This prompts us to ask if the BGP algorithm can be adapted to work with r-wise independent hash functions for small values of r, and if simpler families of hash functions can be used. Indeed, we have found that with $r \geq 2$, an adapted version of the BGP algorithm can be made to generate near-uniform witnesses. We can also bound the probability of failure of the adapted algorithm by a constant. Significantly, the sufficiency of pairwise independence allows us to use computationally efficient xor-based families of hash functions, like $H_{conv}(n, m, 2)$ discussed in Section 2. This provides a significant scaling advantage to our algorithm vis-a-vis the BGP algorithm in practice.

In the context of uniform generation of SAT witnesses, checking if $|R_x| \leq 2n^2$ (line 2 of pseudocode) or if $|R_{x,h,\alpha}| \leq 2n^2$ (line 10 of pseudocode, modified as suggested above) can be done either by approximate model-counting or by repeated invokations of a SAT solver. State-of-the-art approximate model counting techniques [11] rely on randomly sampling the witness space, suggesting a circular dependency. Hence, we choose to use a SAT solver as the back-end engine for enumerating and counting witnesses. Note that if h is chosen randomly from $H_{conv}(n, m, 2)$, the formula for which we seek witnesses is the conjunction of the original (CNF) formula and xor constraints encoding the inclusion of each witness in $h^{-1}(\alpha)$. We therefore choose to use a SAT solver optimized for conjunctions of xor constraints and CNF clauses as the back-end engine; specifically, we use CryptoMiniSAT (version 2.9.2) [1].

Modern SAT solvers often produce partial assignments that specify values of a subset of variables, such that every assignment of values to the remaining variables gives a witness. Since we must find large numbers ($2n^2 \approx 2 \times 10^6$ if $n \approx 1000$) of witnesses, it would be useful to obtain partial assignments from the SAT solver. Unfortunately, conjoining random xor constraints to the original formula reduces the likelihood that large sets of witnesses can be encoded as partial assignments. Thus, each invokation of the SAT solver is likely to generate only a few witnesses, necessitating a large number of calls to the solver. To make matters worse, if the count of witnesses exceeds $2n^2$ and if $i < n - 1$, the check in line 10 of the pseudocode of algorithm BGP (modified as suggested above) fails, and the loop of lines 7–10 iterates once more, requiring generation of up to $2n^2$ witnesses of a modified SAT problem all over again. This can be computationally prohibitive in practice. Indeed, our implementation of the BGP algorithm with CryptoMiniSAT failed to terminate on formulas with few tens of variables, even when running on high-performance computers for 20 hours. This prompts us to ask if the required number of witnesses, or *pivot*, in the BGP algorithm (see line 1 of the pseudocode) can be reduced. We answer this question in the affirmative, and show that the pivot can indeed be reduced to $2n^{1/k}$, where k is an integer ≥ 1. Note that if $k = 3$ and

$n = 1000$, the value of $2n^{1/k}$ is only 20, while $2n^2$ equals 2×10^6. This translates to a significant leap in the sizes of problems for which we can generate random witnesses. There are, however, some practical tradeoffs involved in the choice of k; we defer a discussion of these to a later part of this section.

We now present the UniWit algorithm, which implements the modifications to the BGP algorithm suggested above. UniWit takes as inputs a CNF formula F with n variables, and an integer $k \geq 1$. The algorithm either outputs a witness that is near-uniformly distributed over the space of all witnesses of F or produces a symbol $\perp$ indicating a failed run. We also assume that we have access to a function BoundedSAT that takes as inputs a propositional formula F that is a conjunction of a CNF formula and xor constraints, and an integer $r \geq 0$ and returns a set S of witnesses of F such that $|S| = \min(r, \#F)$, where $\#F$ denotes the count of all witnesses of F.

Algorithm UniWit(F, k)**:**
/* Assume $z_1, \ldots z_n$ are variables in F */
/* Choose a priori the family of hash functions $H(n, m, r), r \geq 2$ to be used */

1: pivot $\leftarrow \lceil 2n^{1/k} \rceil$; $S \leftarrow$ BoundedSAT$(F, \text{pivot} + 1)$;
2: **if** $(|S| \leq \text{pivot})$
3: Let $y_1, \ldots y_{|S|}$ be the elements of S;
4: Choose j at random from $\{1, \ldots |S|\}$ and **return** y_j;
5: **else**
6: $l \leftarrow \lfloor \frac{1}{k} \cdot (\log_2 n) \rfloor$; $i \leftarrow l - 1$;
7: **repeat**
8: $i \leftarrow i + 1$;
9: Choose h at random from $H(n, i - l, r)$;
10: Choose α at random from $\{0, 1\}^{i-l}$;
11: $S \leftarrow$ BoundedSAT$(F \wedge (h(z_1, \ldots z_n) = \alpha), \text{pivot} + 1)$;
12: **until** $(1 \leq |S| \leq \text{pivot})$ or $(i = n)$;
13: **if** $(|S| > \text{pivot})$ or $(|S| < 1)$ **return** $\perp$;
14: **else**
15: Let $y_1, \ldots y_{|S|}$ be the elements of S;
16: Choose j at random from $\{1, \ldots \text{pivot}\}$;.
17: **if** $j \leq |S|$, **return** y_j;
18: **else return** $\perp$;

Implementation Issues: There are four steps in UniWit (lines 4, 9, 10 and 16 of the pseudocode) where random choices are made. In our implementation, in line 10 of the pseudocode, we choose a random hash function from the family $H_{conv}(n, i - l, 2)$, since it is computationally efficient to do so. Recall from Section 2 that choosing a random hash function from $H_{conv}(n, m, 2)$ requires choosing two random bit-vectors. It is straightforward to implement these choices and also the choice of a random $\alpha \in \{0, 1\}^{i-l}$ in line 10 of the pseudocode, if we have access to a source of independent and uniformly distributed random bits. In lines 4 and 16, we must choose a random integer from a specified range.

By using standard techniques (see, for example, the discussion on coin tossing in [5]), this can also be implemented efficiently if we have access to a source of random bits. Since accessing truly random bits is a practical impossibility, our implementation uses pseudorandom sequences of bits generated from nuclear decay processes and available at HotBits [2]. We download and store a sufficiently long sequence of random bits in a file, and access an appropriate number of bits sequentially whenever needed.

In line 11 of the pseudocode for UniWit, we invoke BoundedSAT with arguments $F \wedge (h(z_1, \ldots z_n) = \alpha)$ and pivot $+ 1$. The function BoundedSAT is implemented using CryptoMiniSAT (version 2.9.2), which allows passing a parameter indicating the maximum number of witnesses to be generated. The sub-formula $(h(z_1, \ldots z_n) = \alpha)$ is constructed as follows. As mentioned in Section 2, a random hash function from the family $H_{conv}(n, i-l, 2)$ can be implemented by choosing a random $a \in \{0,1\}^{n+i-l-1}$ and a random $b \in \{0,1\}^{i-l}$. Recalling the definition of h from Section 2, the sub-formula $(h(z_1, \ldots z_n) = \alpha)$ is given by $\bigwedge_{j=1}^{i-l} \left(\left(\bigoplus_{p=1}^{n} (z_p \wedge a[j+p-1]) \oplus b[j] \right) \Leftrightarrow \alpha[j] \right)$.

Analysis of UniWit: Let R_F denote the set of witnesses of the input formula F. Using notation discussed in Section 2, suppose $R_F \subseteq \{0,1\}^n$. For simplicity of exposition, we assume that $\log_2 |R_F| - (1/k) \cdot \log_2 n$ is an integer in the following discussion. A more careful analysis removes this assumption with constant factor reductions in the probability of generation of an arbitrary witness and in the probability of failure of UniWit.

Theorem 3. *Suppose F has n variables and $n > 2^k$. For every witness y of F, the conditional probability that algorithm* UniWit *outputs y on inputs F and k, given that the algorithm succeeds, is bounded below by $\frac{1}{8|R_F|}$.*

Proof. Referring to the pseudocode of UniWit, if $|R_F| \leq 2n^{1/k}$, the theorem holds trivially. Suppose $|R_F| > 2n^{1/k}$, and let Y denote the event that witness y in R_F is output by UniWit on inputs F and k. Let $p_{i,y}$ denote the probability that the loop in lines 7–12 of the pseudocode terminates in iteration i with y in $R_{F,h,\alpha}$, where $\alpha \in \{0,1\}^{i-l}$ is the value chosen in line 10. It follows from the pseudocode that $\mathsf{Pr}[Y] \geq p_{i,y} \cdot (1/2n^{1/k})$, for every $i \in \{l, \ldots n\}$. Let us denote $\log_2 |R_F| - (1/k) \cdot \log_2 n$ by m. Therefore, $2^m \cdot n^{1/k} = |R_F|$. Since $2n^{1/k} < |R_F| \leq 2^n$ and since $l = \lfloor (1/k) \cdot \log_2 n \rfloor$ (see line 6 of pseudocode), we have $l < m + l \leq n$. Consequently, $\mathsf{Pr}[Y] \geq p_{m+l,y} \cdot (1/2n^{1/k})$. The proof is completed by showing that $p_{m+l,y} \geq \frac{1-n^{-1/k}}{2^{m+1}}$. This gives $\mathsf{Pr}[Y] \geq \frac{1-n^{-1/k}}{2^{m+2} \cdot n^{1/k}} = \frac{1-n^{-1/k}}{4|R_F|} \geq \frac{1}{8|R_F|}$, if $n > 2^k$.

To calculate $p_{m+l,y}$, we first note that since $y \in R_F$, the requirement "$y \in R_{F,h,\alpha}$" reduces to "$y \in h^{-1}(\alpha)$". For $\alpha \in \{0,1\}^m$ and $y \in \{0,1\}^n$, we define $q_{m+l,y,\alpha}$ as $\mathsf{Pr}\left[|R_{F,h,\alpha}| \leq 2n^{1/k} \text{ and } h(y) = \alpha : h \xleftarrow{R} H(n,m,r) \right]$, where $r \geq 2$. The proof is now completed by showing that $q_{m+l,y,\alpha} \geq (1 - n^{-1/k})/2^{m+1}$ for every $\alpha \in \{0,1\}^m$ and $y \in \{0,1\}^n$. Towards this end, we define an indicator variable $\gamma_{y,\alpha}$ for every $y \in \{0,1\}^n$ and $\alpha \in \{0,1\}^m$ as follows: $\gamma_{y,\alpha} = 1$ if $h(y) = \alpha$ and $\gamma_{y,\alpha} = 0$ otherwise. Thus, $\gamma_{y,\alpha}$ is a random variable with probability distribution induced by that of h. It is easy to show that (i) $\mathsf{E}[\gamma_{y,\alpha}] = 2^{-m}$, and (ii)

the pairwise independence of h implies pairwise independence of the $\gamma_{y,\alpha}$ variables. We now define $\Gamma_\alpha = \sum_{z \in R_F} \gamma_{z,\alpha}$ and $\mu_{y,\alpha} = \mathsf{E}[\Gamma_\alpha \mid \gamma_{y,\alpha} = 1]$. Clearly, $\Gamma_\alpha = |R_{F,h,\alpha}|$ and $\mu_{y,\alpha} = \mathsf{E}\left[\sum_{z \in R_F} \gamma_{z,\alpha} \mid \gamma_{y,\alpha} = 1\right] = \sum_{z \in R_F} \mathsf{E}[\gamma_{z,\alpha} \mid \gamma_{y,\alpha} = 1]$. Using pairwise independence of the $\gamma_{y,\alpha}$ variables, the above simplifies to $\mu_{y,\alpha} = 2^{-m}(|R_F| - 1) + 1 \leq 2^{-m}|R_F| + 1 = n^{1/k} + 1$. From Markov's inequality, we know that $\mathsf{Pr}[\Gamma_\alpha \leq \kappa \cdot \mu_{y,\alpha} \mid \gamma_{y,\alpha} = 1] \geq 1 - 1/\kappa$ for $\kappa > 0$. With $\kappa = \frac{2}{1+n^{-1/k}}$, this gives $\mathsf{Pr}[|R_{F,h,\alpha}| \leq 2n^{1/k} \mid \gamma_{y,\alpha} = 1] \geq (1 - n^{-1/k})/2$. Since h is chosen at random from $H(n, m, r)$, we also have $\mathsf{Pr}[h(y) = \alpha] = 1/2^m$. It follows that $q_{m+l,y,\alpha} \geq (1 - n^{-1/k})/2^{m+1}$. □

Theorem 4. *Assuming $n > 2^k$, algorithm* UniWit *succeeds (i.e. does not return $\perp$) with probability at least $\frac{1}{8}$.*

Proof. Let P_{succ} denote the probability that a run of algorithm UniWit succeeds. By definition, $P_{succ} = \sum_{y \in R_F} \mathsf{Pr}[Y]$. Using Theorem 3, $P_{succ} \geq \sum_{y \in R_F} \frac{1}{8|R_F|} = \frac{1}{8}$. □

One might be tempted to use large values of the parameter k to keep the value of *pivot* low. However, there are tradeoffs involved in the choice of k. As k increases, the pivot $2n^{1/k}$ reduces, and the chances that BoundedSAT finds more than $2n^{1/k}$ witnesses increases, necessitating further iterations of the loop in lines 7–12 of the pseudocode. Of course, reducing the pivot also means that BoundedSAT has to find fewer witnesses, and each invokation of BoundedSAT is likely to take less time. However, the increase in the number of invokations of BoundedSAT contributes to increased overall time. In our experiments, we have found that choosing k to be either 2 or 3 works well for all our benchmarks (including those containing several thousand variables).

A Heuristic Optimization: A (near-)uniform generator is likely to be invoked a large number of times for the same formula F when generating a set of witnesses of F. If the performance of the generator is sensitive to problem-specific parameter(s) not known a priori, a natural optimization is to estimate values of these parameter(s), perhaps using computationally expensive techniques, in the first few runs of the generator, and then re-use these estimates in subsequent runs on the same problem instance. Of course, this optimization works only if the parameter(s) under consideration can be reasonably estimated from the first few runs. We call this heuristic optimization "leapfrogging".

In the case of algorithm UniWit, the loop in lines 7–12 of the pseudocode starts with i set to $l-1$ and iterates until either i increments to n, or $|R_{F,h,\alpha}|$ becomes no larger than $2n^{1/k}$. For each problem instance F, we propose to estimate a lower bound of the value of i when the loop terminates, from the first few runs of UniWit on F. In all subsequent runs of UniWit on F, we propose to start iterating through the loop with i set to this lower bound. We call this specific heuristic *"leapfrogging i"* in the context of UniWit. Note that leapfrogging may also be used for the parameter s in algorithms XORSample′ and XORSample (see pseudocode of XORSample′). We will discuss more about this in Section 5.

5 Experimental Results

To evaluate the performance of UniWit, we built a prototype implementation and conducted an extensive set of experiments. Since our motivation stems primarily from functional verification, our benchmarks were mostly derived from functional verification of hardware designs. Specifically, we used "bit-blasted" versions of word-level constraints arising from bounded model checking of public-domain and proprietary word-level VHDL designs. In addition, we also used bit-blasted versions of several SMTLib [3] benchmarks of the "QF_BV/bruttomesso/ simple_processor/" category, and benchmarks arising from "Type I" representations of ISCAS'85 circuits, as described in [9].

All our experiments were conducted on a high-performance computing cluster. Each individual experiment was run on a single node of the cluster, and the cluster allowed multiple experiments to run in parallel. Every node in the cluster had two quad-core Intel Xeon processors running at 2.83 GHz with 4 GB of physical memory. We used 3000 seconds as the timeout interval for each invokation of BoundedSAT in UniWit, and 20 hours as the timeout interval for the overall algorithm. If an invokation of BoundedSAT in line 11 of the pseudocode timed out (after 3000 seconds), we repeated the iteration (lines 7–12 of the pseudocode of UniWit) without incrementing i. If the overall algorithm timed out (after 20 hours), we considered the algorithm to have failed. We used either 2 or 3 for the value of the parameter k (see pseudocode of UniWit). This corresponds to restricting the pivot to few tens of witnesses for formulae with a few thousand variables. The exact values of k used for a subset of the benchmarks are indicated in Table 1. A full analysis of the effect of parameter k will require a separate study. As explained earlier, our implementation uses the family $H_{conv}(n, m, 2)$ to select random hash functions in step 9 of the pseudocode.

For purposes of comparison, we also implemented and conducted experiments with algorithms BGP [5], XORSample and XORSample′ [12], using CryptoMiniSAT as the SAT solver in all cases. Algorithm BGP timed out without producing any witness in all but the simplest of cases (involving less than 20 variables). This is primarily because checking whether $|R_{x,h,\alpha}| \leq 2n^2$ for a given $h \in H(n, m, n)$ and *for every* $\alpha \in \{0,1\}^m$, as required in step 10 of algorithm BGP, is computationally prohibitive for values of n and m exceeding few tens. Hence, we do not report any comparison with algorithm BGP. Of the algorithms XORSample and XORSample′, algorithm XORSample′ consistently out-performed algorithm XORSample in terms of both actual time taken and uniformity of generated witnesses. This can be largely attributed to the stringent requirement of algorithm XORSample that its input parameter s must render the model count of the input formula F constrained with s random xor constraints to *exactly* 1. Our experiments indicated that it was extremely difficult to predict or leapfrog the range of values for s such that it met the strict requirement of the model count being *exactly* 1. This forced us to expend significant computing resources to estimate the right value value for s in almost every run, leading to huge performance overheads. Since algorithm XORSample′ consistently outperformed algorithm XORSample, we focus on comparisons with only algorithm XORSample′ in

the subsequent discussion. Note that our benchmarks, when viewed as Boolean circuits, had upto 695 circuit inputs, and 21 of them had more than 95 inputs each. While UniWit and XORSample′ completed execution on all these benchmarks, we could not build ROBDDs for 18 of the above 21 benchmarks within our timeout limit and with 4GB of memory. Hence no comparison with ROBDD-based techniques is reported.

Table 1 presents results of our experiments comparing performance and uniformity of generated witnesses for UniWit and XORSample′ on a subset of benchmarks. The tool and the complete set of results on over 200 benchmarks are available at `http://www.cfdvs.iitb.ac.in/reports/UniWit/`. The first three columns in Table 1 give the name, number of variables and number of clauses of the benchmarks represented as CNF formulae. The columns grouped under UniWit give details of runs of UniWit, while those grouped under XORSample′ give details of runs of XORSample′. For runs of UniWit, the column labeled "k" gives the value of the parameter k used in the corresponding experiment. The column labeled "Range (i)" shows the range of values of i when the loop in lines 7–12 of the pseudocode (see Section 4) terminated in 100 independent runs of the algorithm on the benchmark under consideration. Significantly, this range is uniformly narrow for all our experiments with UniWit. As a result, leapfrogging i is very effective for UniWit.

The column labeled "Run Time" under UniWit in Table 1 gives run times in seconds, separated as $time_1 + time_2$, where $time_1$ gives the average time (over 100 independent runs) to obtain a witness and to identify the lower bound of i for leapfrogging in later runs, while $time_2$ gives the average time to get

Table 1. Performance comparison of UniWit and XORSample′

			UniWit				XORSample′	
Benchmark	#var	Clauses	k	Range (i)	Average Run Time (s)	Var-iance	Average Run Time (s)	Var-iance
case_3_b14	779	2480	2 3	[34,35] [36,37]	49.29+5.27 19.32+1.44	1.58	15061.85+59.31	3.47
case_2_b14	519	1607	3	[38,39]	22.13+2.09	0.57	18005.58+0.73	9.51
case203	214	580	3	[42,44]	16.41+1.04	8.98	18006.85+2.78	230.5
case145	219	558	3	[42,44]	19.84+1.42	1.62	18007.18+2.99	2.32
case14	270	717	2	[44,45]	54.07+2.33	0.65	18004.8+0.9	28.16
case61	289	773	3	[44,46]	30.39+5.49	1.33	18009.1+4.4	11.92
case9	302	821	3	[45,47]	25.64+1.54	2.07	18004.79+0.87	46.15
case10	351	946	2	[60,61]	204.93+17.99	2.68	18008.42+4.85	10.56
case15	319	842	3	[61,63]	91.84+14.64	2.61	18008.34+5.08	11.04
case140	488	1222	3	[99,101]	288.63+23.53	1.41	21214.85+200.64	6.71
squaring14	5397	18141	3	[28,30]	2399.19+1243.81		7089.6+2088.46	
squaring7	5567	18969	3	[26,29]	2358.45+1720.49		4841.4+2340.84	
case39	590	1789	2	[50,50]	710.65+85.22		18159.12+138.22	
case_2_ptb	7621	24889	3	[72,73]	1643.2+225.41		22251.8+177.61	
case_1_ptb	7624	24897	2 3	[70,70] [72,73]	17295.45+454.64 1639.16+219.87		22346.64+204.07	

a solution once we leapfrog i. Our experiments clearly show that leapfrogging i reduces run-times by almost an order of magnitude in most cases. We also report "Run Time" for XORSample′, where times are again reported as $time_1 + time_2$. In this case, $time_1$ gives the average time (over 100 independent runs) taken to find the value of the parameter s in algorithm XORSample′ using a binary search technique, as outlined in a footnote in [12]. As can be seen from Table 1, this is a computationally expensive step, and often exceeds $time_1$ for UniWit by more than two to three orders of magnitude. Once the range of the parameter s is identified from the first 100 independent runs, we use the lower bound of this range to leapfrog s in subsequent runs of XORSample′ on the same problem instance. The values of $time_2$ under "Run Time" for XORSample′ give the average time taken to generate witnesses after leapfrogging s. Note that the difference between $time_2$ values for UniWit and XORSample′ is far less pronounced than the difference between $time_1$ values. In addition, the $time_1$ values for XORSample′ are two to four orders of magnitude larger than the corresponding $time_2$ values, while this factor is almost always less than an order of magnitude for UniWit. Therefore, the total time taken for n_1 runs without leapfrogging, followed by n_2 runs with leapfrogging for XORSample′ far exceeds that for UniWit, even for $n_1 = 100$ and $n_2 \approx 10^6$. This illustrates the significant practical efficiency of UniWit vis-a-vis XORSample′.

Table 1 also reports the scaled statistical variance of relative frequencies of witnesses generated by 5×10^4 runs of the two algorithms on several benchmarks. The scaled statistical variance is computed as $\frac{K}{N-1} \sum_{i=1}^{N} \left(f_i - \left(\frac{\sum_{i=1}^{N} f_i}{N} \right) \right)^2$, where N denotes the number of distinct witnesses generated, f_i denotes the relative frequency of the i^{th} witness, and K (10^{10}) denotes a scaling constant used to facilitate easier comparison. The smaller the scaled variance, the more uniform is the generated distribution. Unfortunately, getting a reliable estimate of the variance requires generating witnesses from runs that sample the witness space sufficiently well. While we could do this for several benchmarks (listed towards the top of Table 1), other benchmarks (listed towards the bottom of Table 1) had too large witness spaces to conduct these experiments within available resources. For those benchmarks where we have variance data, we observe that the variance obtained using XORSample′ is larger (by upto a factor of 43) than those obtained using UniWit in almost all cases. Overall, our experiments indicate that UniWit always works significantly faster and gives more (or comparably) uniformly distributed witnesses vis-a-vis XORSample′ in almost all cases. We also measured the probability of success of UniWit for each benchmark as the ratio of the number of runs for which the algorithm did not return $\perp$ to the total number of runs. We found that this exceeded 0.6 for every benchmark using UniWit.

As an illustration of the difference in uniformity of witnesses generated by UniWit and XORSample′, Figures 1 and 2 depict the frequencies of appearance of various witnesses using these two algorithms for an input CNF formula (case110) with 287 variables and 16,384 satisfying assignments. The horizontal axis in each figure represents witnesses numbered suitably, while the vertical axis represents

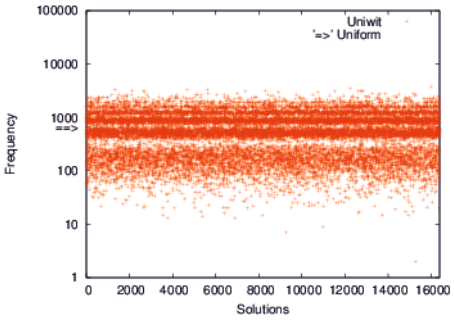


Fig. 1. Sampling by UniWit (k=2)

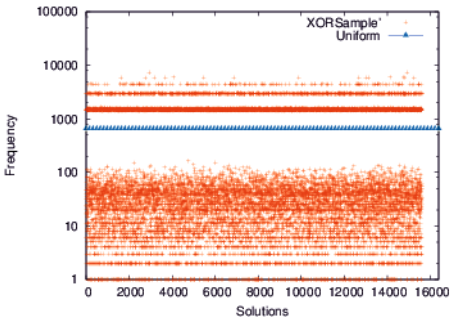


Fig. 2. Sampling by XORSample′

the generated frequencies of witnesses. The frequencies were obtained from 10.8×10^6 successful runs of each algorithm. Interestingly, XORSample′ could find only 15,612 solutions (note the empty vertical band at the right end of Figure 2), while UniWit found all 16,384 solutions. Further, XORSample′ generated each of 15 solutions more than 5,500 times, and more than 250 solutions were generated only once. No such major deviations from uniformity were however observed in the frequencies generated by UniWit. We also found that 15624 out of 16384 (i.e. 95.36%) witnesses generated by UniWit had frequencies in excess of $N_{unif}/8$, where $N_{unif} = 10.8 \times 10^6/16384 \approx 659$. In contrast, only 6047 (i.e. 36.91%) witnesses generated by XORSample′ had frequencies in excess of $N_{unif}/8$.

6 Concluding Remarks

We described UniWit, an algorithm that near-uniformly samples random witnesses of Boolean formulas. We showed that the algorithm scales to reasonably large problems. We also showed that it performs better, in terms of both run time and uniformity, than previous best-of-breed algorithms for this problem. The theoretical guarantees can be further improved with higher independence of the family of hash functions used in UniWit (see `http://www.cfdvs.iitb.ac.in/reports/UniWit` for details).

We are yet to fully explore the parameter space and the effect of pseudorandom generators other than HotBits for UniWit. There is a trade off between failure probability, time for first witness, and time for subsequent witnesses. During our experiments, we observed the acute dearth of benchmarks available in the public domain for this important problem. We hope that our work will lead to development of benchmarks for this problem. Our focus here has been on Boolean constraints, which play a prominent role in hardware design. Extending the algorithm to handle user-provided biases would be an interesting direction of future work. Yet another interesting extension would be to consider richer constraint languages and build a uniform generator of witnesses modulo theories, leveraging recent progress in satisfiability modulo theories, c.f., [10].

References

1. CryptoMiniSAT, http://www.msoos.org/cryptominisat2/
2. HotBits, http://www.fourmilab.ch/hotbits
3. SMTLib, http://goedel.cs.uiowa.edu/smtlib/
4. Bacchus, F., Dalmao, S., Pitassi, T.: Algorithms and complexity results for #SAT and Bayesian inference. In: Proc. of FOCS, pp. 340–351 (2003)
5. Bellare, M., Goldreich, O., Petrank, E.: Uniform generation of NP-witnesses using an NP-oracle. Information and Computation 163(2), 510–526 (1998)
6. Bentley, B.: Validating a modern microprocessor. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 2–4. Springer, Heidelberg (2005)
7. Chandra, A.K., Iyengar, V.S.: Constraint solving for test case generation: A technique for high-level design verification. In: Proc. of ICCD, pp. 245–248 (1992)
8. Chang, K., Markov, I.L., Bertacco, V.: Functional Design Errors in Digital Circuits: Diagnosis Correction and Repair. Springer (2008)
9. Darwiche, A.: A compiler for deterministic, decomposable negation normal form. In: Proc. of AAAI, pp. 627–634 (2002)
10. de Moura, L.M., Bjørner, N.: Satisfiability Modulo Theories: Introduction and Applications. Commun. ACM 54(9), 69–77 (2011)
11. Gomes, C.P., Sabharwal, A., Selman, B.: Model counting: A new strategy for obtaining good bounds. In: Proc. of AAAI, pp. 54–61 (2006)
12. Gomes, C.P., Sabharwal, A., Selman, B.: Near-Uniform sampling of combinatorial spaces using XOR constraints. In: Proc. of NIPS, pp. 670–676 (2007)
13. Guralnik, E., Aharoni, M., Birnbaum, A.J., Koyfman, A.: Simulation-based verification of floating-point division. IEEE Trans. on Computers 60(2), 176–188 (2011)
14. Jerrum, M.R., Valiant, L.G., Vazirani, V.V.: Random generation of combinatorial structures from a uniform distribution. Theoretical Computer Science 43(2-3), 169–188 (1986)
15. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. Science 220(4598), 671–680 (1983)
16. Kitchen, N., Kuehlmann, A.: Stimulus generation for constrained random simulation. In: Proc. of ICCAD, pp. 258–265 (2007)
17. Mansour, Y., Nisan, N., Tiwari, P.: The computational complexity of universal hashing. Theoretical Computer Science 107(1), 235–243 (2002)
18. Naveh, Y., Rimon, M., Jaeger, I., Katz, Y., Vinov, M., Marcus, E., Shurek, G.: Constraint-based random stimuli generation for hardware verification. In: Proc. of AAAI, pp. 1720–1727 (2006)
19. Roth, D.: On the hardness of approximate reasoning. Artificial Intelligence 82(1), 273–302 (1996)
20. Sipser, M.: A complexity theoretic approach to randomness. In: Proc. of STOC, pp. 330–335 (1983)
21. Stockmeyer, L.: The complexity of approximate counting. In: Proc. of STOC, pp. 118–126 (1983)
22. Wei, W., Erenrich, J., Selman, B.: Towards efficient sampling: Exploiting random walk strategies. In: Proc. of AAAI, pp. 670–676 (2004)
23. Yuan, J., Aziz, A., Pixley, C., Albin, K.: Simplifying boolean constraint solving for random simulation-vector generation. IEEE Trans. on CAD of Integrated Circuits and Systems 23(3), 412–420 (2004)

Equivalence of Extended Symbolic Finite Transducers*

Loris D'Antoni[1] and Margus Veanes[2]

[1] University of Pennsylvania
lorisdan@cis.upenn.edu
[2] Microsoft Research
margus@microsoft.com

Abstract. Symbolic Finite Transducers augment classic transducers with symbolic alphabets represented as parametric theories. Such extension enables succinctness and the use of potentially infinite alphabets while preserving closure and decidability properties. Extended Symbolic Finite Transducers further extend these objects by allowing transitions to read consecutive input elements in a single step. While when the alphabet is finite this extension does not add expressiveness, it does so when the alphabet is symbolic. We show how such increase in expressiveness causes decision problems such as equivalence to become undecidable and closure properties such as composition to stop holding. We also investigate how the automata counterpart, Extended Symbolic Finite Automata, differs from Symbolic Finite Automata. We then introduce the subclass of Cartesian Extended Symbolic Finite Transducers in which guards are limited to conjunctions of unary predicates. Our main result is an equivalence algorithm for such subclass in the single-valued case. Finally, we model real world problems with Cartesian Extended Symbolic Finite Transducers and use the equivalence algorithm to prove their correctness.

1 Introduction

Finite automata have proven to be an effective tool in a wide range of applications, from regular expressions to network packet inspection [15]. Finite transducers extend finite automata with outputs and can model functions from strings to strings such as natural language transformations [12]. Due to their closure and decidability properties, these models are widely used in practice but they have three major disadvantages: 1) their number of transitions usually "blows up" when dealing with large alphabets; 2) they cannot model infinite alphabets; and 3) transitions cannot express relations between adjacent input symbols.

Symbolic Finite Automata/Transducers [16] or SFAs/SFTs respectively, are an extension of traditional automata and transducers that attempts to solve problems 1 and 2 above by allowing transitions to be labelled with arbitrary

* Loris D'Antoni's research was partially supported by NSF Expeditions in Computing award CCF 1138996.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 624–639, 2013.

predicates in a specified theory. When such theory is decidable SFAs and SFTs enjoy the same properties of finite automata and transducers, such as closure under composition and decidability of equivalence (for single-valued SFTs). In [16], Symbolic Transducers or STs (SFTs with registers) are proposed in order to cope with the third problem above. STs are however undecidable with respect to most analysis problems, even emptiness.

In our previous work on the topic of analysis of string coders [4], we introduce *Extended Symbolic Finite Automata/Transducers* or ESFAs/ESFTs, that add finite lookahead to SFAs/SFTs. This extension allows to read multiple input symbols in a single transition and combine their values in the output. ESFTs can be viewed as a subclass of STs with a restricted use of registers that mimic "look-behind". This view is used in [4] to map ESFTs directly to STs in order to overcome the problem that ESFTs are not closed under composition. In other words, it addresses the composition problem by first converting ESFTs to STs, then composing the STs, and finally converting the result back into an ESFT using a semi-decision procedure. The formal properties of ESFTs have not been fully understood yet. From the point of view of analysis, the key operations that are desired are *composition* and *equivalence* (for single-valued ESFTs). Then, for example, the functional correctness of a string (*encoder*,*decoder*) pair (E, D) (for example UTF8 to UTF16 encoding) can be decided by checking the equivalence of $\lambda x.D(E(x))$ with $\lambda x.x$. Other properties, such as commutativity and idempotence, also depend on composition and equivalence.

The topic that is left open in [4] is decidability of equivalence checking of ESFTs. Our main theoretical contribution in this paper is a complete classification, in terms of guard complexity and lookahead, of the cases in which the equivalence problem is decidable for ESFTs. We first show that one-equality or equivalence in the single-valued case is in general undecidable, contrasting the finite alphabet setting where lookahead does not matter [18, Theorem 2.17]. We then introduce the notion of Cartesian ESFT, in which transition guards are constrained to be conjunctions of unary predicates, and show that one-equality and equivalence are decidable for single-valued Cartesian ESFTs. This is a proper extension of the decidability result of one-equality of SFTs [16]. The key tool that we need to prove the result is Lemma 2.

We also analyze basic properties of ESFAs and show how they differ from SFAs. We prove ESFAs to be not closed under intersection and show that equivalence and universality of ESFAs are both undecidable problems.

Applications. We present four applications of our models in different areas. We first extend the result of [4] by proving the correctness of four real world string encoders. The new equivalence algorithm is a full decision procedure for the Cartesian case, unlike the semi-decision procedure in [4] that may fail to terminate in some incorrect instances. Our second and third applications are in the context of networking and present new classes of programs that can be modelled as ESFAs/ESFTs. We show how 1) ESFAs can be used for the task of deep-packet inspection, and 2) ESFTs can succinctly represent transformations

between headers of different network protocols. Our fourth case study shows the use of additional theories for the analysis of list manipulating programs.

Contributions. In summary, we offer the following contributions:

- we study the closure and decidability properties of ESFAs (Section 3.1);
- we study the equivalence problem for ESFTs (Section 3.2):
  - we prove the equivalence of single-valued ESFTs to be undecidable;
  - we present a novel algorithm for the equivalence of single-valued Cartesian ESFTs;
- we extend the negative result on ESFTs composition presented in [4] (Section 3.3); and
- we analyze the performance of the equivalence algorithm for "Cartesian" ESFTs on real examples and propose new applications for ESFAs and ESFTs (Section 4).

We finally summarize previous work and conclude (Section 5 and 6).

2 Extended Symbolic Finite Transducers

We assume a recursively enumerable (r.e.) *background universe* $\mathcal{U}$ with built-in function and relation symbols. Definitions below are given with $\mathcal{U}$ as an implicit parameter. We use λ-expressions for representing anonymous functions that we call λ-*terms*. A Boolean λ-term $\lambda x.\varphi(x)$, where x is a variable of type σ is called a σ-predicate. Our notational conventions are consistent with the definition of symbolic transducers [16]. The universe is multi-typed with $\mathcal{U}^\tau$ denoting the sub-universe of elements of type τ. We write Σ for $\mathcal{U}^\sigma$ and Γ for $\mathcal{U}^\gamma$.

A *label theory* is given by a recursively enumerable set Ψ of formulas that is closed under Boolean operations, substitution, equality and if-then-else terms. A label theory Ψ is *decidable* when satisfiability for $\varphi \in \Psi$, $IsSat(\varphi)$, is decidable.

For σ-predicates φ, we assume an effective *witness* function $\mathscr{W}$ such that, if $IsSat(\varphi)$ then $\mathscr{W}(\varphi) \in [\![\varphi]\!]$, where $[\![\varphi]\!] \subseteq \mathcal{U}^\sigma$ is the set of all values that satisfy φ; φ is *valid*, $IsValid(\varphi)$, when $[\![\varphi]\!] = \mathcal{U}^\sigma$.

We are studying in this paper an extension of SFTs with *lookahead*, called *extended* SFTs or *ESFTs*. Originally, ESFTs were introduced in [4] for the purposes of analyzing string encoders and decoders, where a semi-decision procedure was provided for converting STs (SFTs with registers) into ESFTs.

Definition 1. An *Extended Symbolic Finite Transducer (ESFT)* with *input type* σ and *output type* γ is a tuple $A = (Q, q^0, R)$,

- Q is a finite set of *states*;
- $q^0 \in Q$ is the *initial state*;
- R is a finite set of *rules*, $R = \Delta \cup F$, where
- Δ is a set of *transitions* $r = (p, \ell, \varphi, f, q)$, denoted $p \xrightarrow[\ell]{\varphi/f} q$, where
  $p \in Q$ is the *start* state of r;

$\ell \geq 1$ is the *lookahead* of r;
φ, the *guard* of r, is a σ^ℓ-predicate;
f, the *output* of r, is a $(\sigma^\ell \to \gamma)$-sequence;
$q \in Q$ is the *continuation* state of r.

– F is a set of *finalizers* $r = (p, \ell, \varphi, f)$, denoted $p \xrightarrow[\ell]{\varphi/f} \bullet$, with components as above and where ℓ may be 0.

The *lookahead* of A is the maximum of all lookaheads of rules in R. An ESFT all of whose rules have output [] is an *Extended Symbolic Finite Automaton* (*ESFA*).

A finalizer is a rule without a continuation state. A finalizer with lookahead ℓ is used when the end of the input sequence has been reached with *exactly* ℓ input elements remaining. A finalizer is a generalization of a final state. In a classical setting, finalizers can be avoided by adding a new symbol to the alphabet that is only used to mark the end of the input. In the presence of arbitrary input types, this is not always possible without affecting the theory, e.g., when the input type is $\mathbb{Z}$ then that symbol would have to be outside $\mathbb{Z}$.

In the sequel let $A = (Q, q^0, R)$, $R = \Delta \cup F$, be a fixed ESFT with input type σ and output type γ. The semantics of rules in R is as follows:

$$[\![p \xrightarrow[\ell]{\varphi/f} q]\!] \stackrel{\text{def}}{=} \{p \xrightarrow{[a_0,\ldots,a_{\ell-1}]/[\![f]\!](a_0,\ldots,a_{\ell-1})} q \mid (a_0, \ldots, a_{\ell-1}) \in [\![\varphi]\!]\}$$

We write $s_1 \cdot s_2$ for the concatenation of two sequences s_1 and s_2.

Definition 2. For $u \in \Sigma^*, v \in \Gamma^*, q \in Q, q' \in Q \cup \{\bullet\}$, define $q \xrightarrow{u/v}\!\!\!\!\!\rightarrow_A q'$ as follows: there exists $n \geq 0$ and $\{p_i \xrightarrow{u_i/v_i} p_{i+1} \mid i \leq n\} \subseteq [\![R]\!]$ such that

$$u = u_0 \cdot u_1 \cdots u_n, \quad v = v_0 \cdot v_1 \cdots v_n, \quad q = p_0, \quad q' = p_{n+1}.$$

Let also $q \xrightarrow{[]/[]}\!\!\!\!\!\rightarrow_A q$ for all $q \in Q_A$.

Definition 3. The *transduction of A*, $\mathscr{T}_A(u) \stackrel{\text{def}}{=} \{v \mid q^0 \xrightarrow{u/v}\!\!\!\!\!\rightarrow \bullet\}$.

The following example represents typical (realistic) ESFTs over a label theory of linear modular arithmetic. We use the following abbreviated notation for rules, by omitting explicit λ's. We write

$$p \xrightarrow[\ell]{\varphi(\bar{x})/[f_1(\bar{x}),\ldots,f_k(\bar{x})]} q \quad \text{for} \quad p \xrightarrow[\ell]{\lambda\bar{x}.\varphi(\bar{x})/\lambda\bar{x}.[f_1(\bar{x}),\ldots,f_k(\bar{x})]} q,$$

where φ and f_i are terms whose free variables are among $\bar{x} = (x_0, \ldots, x_{\ell-1})$.

Example 1. The example illustrates the standard encoding BASE64, that is used to transfer binary data in textual format, e.g., in emails via the protocol MIME. The digits of the encoding are chosen in the safe ASCII range of characters that remain unmodified during transport over textual media. Assume that the input

type and the output type are both BYTE, that is the set of integers between 0 and 255. *Base64encode* is an ESFT with one state and four rules:

$$p \xrightarrow[3]{true/[\ulcorner b_2^7(x_0)\urcorner, \ulcorner (b_0^1(x_0)\ll 4)|b_4^7(x_1)\urcorner, \ulcorner (b_0^3(x_1)\ll 2)|b_6^7(x_2)\urcorner, \ulcorner b_0^5(x_2)\urcorner]} p$$
$$p \xrightarrow[0]{true/[]} \bullet \qquad p \xrightarrow[1]{true/[\ulcorner b_2^7(x_0)\urcorner, \ulcorner b_0^1(x_0)\ll 4\urcorner, \texttt{'='}, \texttt{'='}]} \bullet$$
$$p \xrightarrow[2]{true/[\ulcorner b_2^7(x_0)\urcorner, \ulcorner (b_0^1(x_0)\ll 4)|b_4^7(x_1)\urcorner, \ulcorner b_0^3(x_1)\ll 2\urcorner, \texttt{'='}]} \bullet$$

where $b_n^m(x)$ extracts bits m through n from x, e.g., $b_2^3(13) = 3$, $x|y$ is bitwise OR of x and y, $x\ll k$ is x shifted left by k bits, and $\ulcorner x \urcorner$ is the mapping

$$\ulcorner x \urcorner \stackrel{\text{def}}{=} (x{\leq}25\,?\,x{+}65 : (x{\leq}51\,?\,x{+}71 : (x{\leq}61\,?\,x{-}4 : (x{=}62\,?\,\texttt{'+'} : \texttt{'/'}))))$$

of values between 0 and 63 into a standardized sequence of safe ASCII character codes. The last two finalizers correspond to the cases when the length of the input sequence is not a multiple of three. Observe that the length of the output sequence is always a multiple of four. The character `'='` (61 in ASCII) is used as a padding character and it is not a BASE64 digit. i.e., `'='` is not in the range of $\ulcorner x \urcorner$.

Base64decode in an ESFT that decodes a BASE64 encoded sequence back into the original byte sequence. *Base64decode* has also one state and four rules:

$$q \xrightarrow[4]{\bigwedge_{i=0}^{3} \beta_{64}(x_i)/[(\llcorner x_0 \lrcorner \ll 2)|b_4^5(\llcorner x_1 \lrcorner), (b_0^3(\llcorner x_1 \lrcorner)\ll 4)|b_2^5(\llcorner x_2 \lrcorner), (b_0^1(\llcorner x_2 \lrcorner)\ll 6)|\llcorner x_3 \lrcorner]} q$$
$$q \xrightarrow[0]{true/[]} \bullet \qquad q \xrightarrow[4]{\beta_{64}(x_0)\wedge\beta'_{64}(x_1)\wedge x_2=\texttt{'='}\wedge x_3=\texttt{'='}/[(\llcorner x_0 \lrcorner \ll 2)|b_4^5(\llcorner x_1 \lrcorner)]} \bullet$$
$$q \xrightarrow[4]{\beta_{64}(x_0)\wedge\beta_{64}(x_1)\wedge\beta''_{64}(x_2)\wedge x_3=\texttt{'='}/[(\llcorner x_0 \lrcorner \ll 2)|b_4^5(\llcorner x_1 \lrcorner), (b_0^3(\llcorner x_1 \lrcorner)\ll 4)|b_2^5(\llcorner x_2 \lrcorner)]} \bullet$$

The function $\llcorner y \lrcorner$ is the inverse of $\ulcorner x \urcorner$, i.e., $\llcorner \ulcorner x \urcorner \lrcorner = x$, for $0 \leq x \leq 63$. The predicate $\beta_{64}(y)$ is true iff y is a valid BASE64 digit, i.e., $y = \ulcorner x \urcorner$ for some x, $0 \leq x \leq 63$. The predicates $\beta'_{64}(y)$ and $\beta''_{64}(y)$ are restricted versions of $\beta_{64}(y)$. Unlike *Base64encode*, *Base64decode* does not accept all input sequences of bytes, and sequences that do not correspond to any encoding are rejected.[1] ⊠

The following subclass of ESFTs captures transductions that behave as partial functions from Σ^* to Γ^*.

Definition 4. A function $\mathbf{f} : X \to 2^Y$ is *single-valued* if $|\mathbf{f}(x)| \leq 1$ for all $x \in X$. An ESFT A is *single-valued* if $\mathscr{T}_A$ is single-valued.

A sufficient condition for single-valuedness is determinism. We define $\varphi \curlywedge \psi$, where φ is a σ^m-predicate and ψ a σ^n-predicate, as the $\sigma^{\max(m,n)}$-predicate $\lambda(x_1, \ldots, x_{\max(m,n)}).\varphi(x_1, \ldots, x_m) \wedge \psi(x_1, \ldots, x_n)$. We define *equivalence of f and g modulo φ*, $f \equiv_\varphi g$, as: $IsValid(\lambda\bar{x}.(\varphi(\bar{x}) \Rightarrow f(\bar{x}) = g(\bar{x})))$.

Definition 5. A is *deterministic* if for all $p \xrightarrow[\ell]{\varphi/f} q, p \xrightarrow[\ell']{\varphi'/f'} q' \in R$:

[1] For more information see `http://www.rise4fun.com/Bek/tutorial/base64`.

(a) Assume $q, q' \in Q$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $q = q'$, $\ell = \ell'$ and $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(b) Assume $q = q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ and $\ell = \ell'$ then $f \equiv_{\varphi \curlywedge \varphi'} f'$.
(c) Assume $q \in Q$ and $q' = \bullet$. If $\mathit{IsSat}(\varphi \curlywedge \varphi')$ then $\ell > \ell'$.

Intuitively, determinism means that no two rules may overlap. It follows from the definitions that if A is deterministic then A is single-valued. Both ESFTs in Example 1 are deterministic.

The *domain* of a function $\mathbf{f} : X \to 2^Y$ is $\mathscr{D}(\mathbf{f}) \stackrel{\text{def}}{=} \{x \in X \mid \mathbf{f}(x) \neq \emptyset\}$ and for an ESFT A, $\mathscr{D}(A) \stackrel{\text{def}}{=} \mathscr{D}(\mathscr{T}_A)$. When A is single-valued, and $u \in \mathscr{D}(A)$, we treat A as a partial function from Σ^* to Γ^* and write $A(u)$ for the value v such that $\mathscr{T}_A(u) = \{v\}$. For example, $\mathit{Base64encode}(\texttt{"Foo"}) = \texttt{"Rm9v"}$ and $\mathit{Base64decode}(\texttt{"QmFy"}) = \texttt{"Bar"}$.

Cartesian ESFTs. We introduce a subclass of ESFTs that plays an important role in this paper. A binary relation R over X is *Cartesian over* X if R is the Cartesian product $R_1 \times R_2$ of some $R_1, R_2 \subseteq X$. The definition is lifted to n-ary relations and σ^n-predicates for $n \geq 2$ in the obvious way. In order to decide if a satisfiable σ^n-predicate φ is Cartesian over σ, let $(a_0, \ldots, a_{n-1}) = \mathscr{W}(\varphi)$ and perform the following validity check:

$$\mathit{IsCartesian}(\varphi) \stackrel{\text{def}}{=} \forall \bar{x}\,(\varphi(\bar{x}) \Leftrightarrow \bigwedge_{i<n} \varphi(a_0, \ldots, a_{i-1}, x_i, a_{i+1}, \ldots, a_{n-1}))$$

In other words, a σ^n-predicate φ is Cartesian over σ if φ can be rewritten equivalently as a conjunction of n independent σ-predicates.

Definition 6. An ESFT (ESFA) is *Cartesian* if all its guards are Cartesian.

Both ESFTs in Example 1 are Cartesian. *Base64encode* trivially so, while the guards of all rules of *Base64decode* are conjunctions of independent unary predicates. In contrast, a predicate such as $\lambda(x_0, x_1).x_0 = x_1$ is not Cartesian.

Note that $\mathit{IsCartesian}(\varphi)$ is decidable by using the decision procedure of the label theory. Namely, decide unsatisfiability of $\neg \mathit{IsCartesian}(\varphi)$.

3 ESFAs and ESFTs Properties

We prove some basic properties of ESFAs and ESFTs and show how they drastically differ from SFAs and SFTs. First, we investigate basic ESFAs properties. Secondly, we prove the undecidability of ESFT equivalence and propose a new one-equality algorithm for the subclass of Cartesian ESFTs. Finally, we present some preliminary results on ESFT composition.

3.1 ESFAs Properties

In this section, we analyse closure and decidability properties of Extended Symbolic Finite Automata. We show how ESFAs have properties similar to those of context free grammars rather than regular languages. First, we show how checking the emptiness of the intersection of two ESFA definable languages is an undecidable problem.

Theorem 1 (Domain Intersection). *Given two ESFAs A and B with lookahead 2 over quantifier free successor arithmetic and tuples, checking whether there exists an input accepted by both A and B is undecidable.*

While checking the emptiness of an ESFA is a decidable problem, it is not possible to decide whether an ESFA accepts every possible input. It follows that equivalence is also undecidable.

Theorem 2 (Emptiness, Universality and Equivalence). *Given an ESFA A and B over σ checking whether A does not accept any input is decidable while, checking whether A accepts all the sequences in σ^* or whether A and B accept the same language are both undecidable problems.*

Combining Theorems 1 and 2 with a simple construction we obtain the following closure properties.

Theorem 3 (Closure Properties). *ESFAs are closed under union, but they are not closed under complement and intersection.*

Finally, Cartesian ESFAs capture exactly the class of SFA definable languages.

Theorem 4 (Cartesian ESFA iff SFA). *SFAs and Cartesian ESFAs are equivalent in expressiveness.*

From Theorem 4 we have that Cartesian ESFAs enjoy all the properties of SFAs (regular languages) such as boolean closures and decidability of equivalence.

3.2 Equivalence of ESFTs

While the general equivalence problem of $\mathscr{T}_A = \mathscr{T}_B$ is already undecidable for very restricted classes of finite state transducers [6], the problem is decidable for SFTs in the single-valued case. More generally, one-equality of transductions (defined next) is decidable for SFTs (over decidable label theories).

Definition 7. Functions $\mathbf{f}, \mathbf{g} : X \to 2^Y$ are *one-equal*, $\mathbf{f} \overset{1}{=} \mathbf{g}$, if for all $x \in X$, if $x \in \mathscr{D}(\mathbf{f}) \cap \mathscr{D}(\mathbf{g})$ then $|\mathbf{f}(x) \cup \mathbf{g}(x)| = 1$. Let

$$\mathbf{f} \uplus \mathbf{g}(x) \overset{\text{def}}{=} \begin{cases} \mathbf{f}(x) \cup \mathbf{g}(x), & \text{if } x \in \mathscr{D}(\mathbf{f}) \cap \mathscr{D}(\mathbf{g}); \\ \emptyset, & \text{otherwise.} \end{cases}$$

Proposition 1. $\mathbf{f} \overset{1}{=} \mathbf{g}$ *iff* $\mathbf{f} \uplus \mathbf{g}$ *is single-valued.*

Note that $\mathbf{f} \overset{1}{=} \mathbf{f}$ iff $\mathbf{f}$ is single-valued. Thus, one-equality is a more refined notion than single-valuedness, because an effective construction of $A \uplus B$ such that $\mathscr{T}_{A \uplus B} = \mathscr{T}_A \uplus \mathscr{T}_B$ may not always be feasible or even possible for some classes of transducers.

Definition 8. Functions $\mathbf{f}, \mathbf{g} : X \to 2^Y$ are *domain-equivalent* if $\mathscr{D}(\mathbf{f}) = \mathscr{D}(\mathbf{g})$.

Definitions 7 and 8 are lifted to (E)SFTs. For domain-equivalent single-valued transducers A and B, $A \overset{1}{=} B$ implies equivalence of A and B ($\mathscr{T}_A = \mathscr{T}_B$).

A natural question that arises is whether decidability of one-equality of SFTs generalizes to ESFTs. The answer is positive for the subclass of *Cartesian* ESFTs (that includes ESFTs in Example 1), but negative in general. We first show that one-equality of ESFTs over decidable label theories is undecidable in general.

Theorem 5 (One-Equality). *One-equality of ESFTs with lookahead 2, over quantifier free successor arithmetic and tuples is undecidable.*

Proof. We give a reduction from the Domain Intersection problem of Theorem 1. Let A_1 and A_2 be ESFAs with lookahead 2 over quantifier free successor arithmetic and tuples. We construct ESFTs A'_i, for $i \in \{1, 2\}$, as follows:

$$A'_i = (Q_{A_i}, q^0_{A_i}, \Delta_{A_i} \cup \{p \xrightarrow[k]{\varphi/[i]} \bullet \mid p \xrightarrow[k]{\varphi} \bullet \in F_{A_i}\})$$

So $\mathscr{T}_{A'_i}(t) = \{[i]\}$ if $t \in \mathscr{D}(A_i)$ and $\mathscr{T}_{A'_i}(t) = \emptyset$ otherwise. Let $\mathbf{f} = \mathscr{T}_{A'_1} \uplus \mathscr{T}_{A'_2}$. So

- $|\mathbf{f}(t)| = 0$ iff $t \notin \mathscr{D}(A_1) \cup \mathscr{D}(A_2)$;
- $|\mathbf{f}(t)| = 1$ iff $t \in \mathscr{D}(A_1) \cup \mathscr{D}(A_2)$ and $t \notin \mathscr{D}(A_1) \cap \mathscr{D}(A_2)$;
- $|\mathbf{f}(t)| = 2$ iff $t \in \mathscr{D}(A_1) \cap \mathscr{D}(A_2)$.

It follows that $A'_1 \overset{1}{=} A'_2$ iff (by Proposition 1) $\mathbf{f}$ is single-valued iff $\mathscr{D}(A_1) \cap \mathscr{D}(A_2) = \emptyset$. Now use Theorem 1. ⊠

The main decidability result of the paper is Theorem 6 that extends the corresponding result for SFTs [16, Theorem 1]. We use the following definitions. A transition, $p \xrightarrow[\ell]{\varphi/f} q$ where $\ell > 1$, φ is Cartesian and $\mathscr{W}(\varphi) = (a_1, \ldots, a_\ell)$, is represented, given $\varphi_i = \lambda x.\varphi(a_1, \ldots, a_{i-1}, x, a_{i+1}, \ldots, a_\ell)$, by the following path of *split* transitions,

$$p \xrightarrow[1]{\varphi_1/f} p_1 \xrightarrow[1]{\varphi_2/\perp} p_2 \cdots p_{\ell-1} \xrightarrow[1]{\varphi_\ell/\perp} q$$

where p_i for $1 \leq i < \ell$ are new *temporary* states, and the output f is postponed until all input elements have been read. Let Δ^{s}_A denote such *split* view of Δ_A. Here we assume that all finalizers have lookahead zero, since we do not assume ESFTs here to be deterministic.

Example 2. It is trivial to transform any ESFT into an equivalent (possibly nondeterministic) form where all finalizers have zero lookahead. Consider the ESFT *Base64encode* in Example 1. In the last two finalizers, replace $\bullet$ with a new state p_1 and and add the new finalizer $p_1 \xrightarrow[0]{true/[]} \bullet$. ⊠

Definition 9. Let A and B be Cartesian ESFTs with same input and output types and zero-lookahead finalizers. The *product* of A and B is the following

product ESFT $A{\times}B$. The *initial state* $q^0_{A\times B}$ of $A{\times}B$ is (q^0_A, q^0_B). The states and transitions of $A{\times}B$ are obtained as the least fixed point of

$$\left.\begin{array}{c}(p,q)\in Q_{A\times B}\\ p \xrightarrow[1]{\varphi/f} p' \in \Delta^{\mathrm{s}}_A\\ q \xrightarrow[1]{\psi/g} q' \in \Delta^{\mathrm{s}}_B\end{array}\right\} \overset{IsSat(\varphi\wedge\psi)}{\Longrightarrow} (p',q')\in Q_{A\times B}, \quad (p,q) \xrightarrow[1]{\varphi\wedge\psi/(f,g)} (p',q') \in \Delta_{A\times B}$$

Let $F_{A\times B}$ be the set of all rules $(p,q) \xrightarrow[0]{true/(v,w)} \bullet$ such that $p \xrightarrow[0]{true/v} \bullet \in F_A$, $q \xrightarrow[0]{true/w} \bullet \in F_B$, and $(p,q) \in Q_{A\times B}$. Finally, remove from $Q_{A\times B}$ (and $\Delta_{A\times B}$) all dead ends (non-initial states from which $\bullet$ is not reachable).

We lift the definition of transductions to product ESFTs. A pair-state $(p,q) \in Q_{A\times B}$ is *aligned* if all transitions from (p,q) have outputs (f,g) such that $f \neq \bot$ and $g \neq \bot$. The relation $\xrightarrow{\!\!/\!\!\!\gg}_{A\times B}$ is defined analogously to ESFTs.

Lemma 1 (Product). *For all aligned* $(p,q) \in Q_{A\times B}$, $u \in \Sigma^*$, $v, w \in \Gamma^*$:

$$(p,q) \xrightarrow{u/(v,w)}\!\!\!\!\gg_{A\times B} \bullet \quad \Leftrightarrow \quad p \xrightarrow{u/v}\!\!\!\!\gg_A \bullet \wedge q \xrightarrow{u/w}\!\!\!\!\gg_B \bullet.$$

We define also, for all $u \in \Sigma^*$, $\mathscr{T}_{A\times B}(u) \overset{\text{def}}{=} \{(v,w) \mid q^0_{A\times B} \xrightarrow{u/(v,w)}\!\!\!\!\gg \bullet\}$ and $\mathscr{D}(A{\times}B) \overset{\text{def}}{=} \mathscr{D}(\mathscr{T}_{A\times B})$. Lemma 1 implies that $\mathscr{D}(A{\times}B) = \mathscr{D}(A) \cap \mathscr{D}(B)$ and $A \overset{1}{\neq} B$ iff there exists u and $v \neq w$ such that $(v,w) \in \mathscr{T}_{A\times B}(u)$.

Next we prove an *alignment* lemma that allows us to either effectively eliminate all non-aligned pair-states from $A{\times}B$ without affecting $\mathscr{T}_{A\times B}$ or else to establish that $A \overset{1}{\neq} B$. A product ESFT is *aligned* if all pair-states in it are aligned.

Lemma 2 (Alignment). *If* $A \overset{1}{=} B$ *then there exists an aligned product ESFT that is equivalent to* $A{\times}B$. *Moreover, there is an effective procedure that either constructs it or else proves that* $A \overset{1}{\neq} B$, *if the label theory is decidable.*

Proof. The product $A{\times}B$ is incrementally transformed by eliminating non aligned pair-states from it. Each iteration preserves equivalence. Using DFS, initialize the search *frontier* to be $\{q^0_{A\times B}\}$. Pick (and remove) a state (p,q) from the frontier and consider all transitions starting from it. The main two cases are the following:

1. If there are transitions from (p,q) where both the A-output f and the B-output g are $(\sigma^\ell \to \gamma)$-sequences with equal lookahead (say $\ell = 2$):

$$(p,q) \xrightarrow[1]{\varphi/(f,g)} (p_1,q_1) \xrightarrow[1]{\psi/(\bot,\bot)} (p_2,q_2)$$

replace the path with the following combined transition with lookahead 2

$$(p,q) \xrightarrow[2]{\lambda(x_0,x_1).\varphi(x_0)\wedge\psi(x_1)/(f,g)} (p_2,q_2).$$

and add (p_2, q_2) to the frontier unless (p_2, q_2) has already been visited. Note that $(p_2, q_2) \in Q_A \times Q_B$ and thus (p_2, q_2) is aligned.

2. Assume there are transitions where the A-output f is a $(\sigma^k \to \gamma)$-sequence and the B-output g is a $(\sigma^\ell \to \gamma)$-sequence ($k \neq \ell$, say $k = 2$ and $\ell = 1$):

$$(p, q) \xrightarrow{\varphi/(f,g)} (p_1, q_1) \xrightarrow{\psi/(\bot,g_1)} (p_2, q_2)$$

So p_1 is temporary while q_1 is not.
Decide if f can be split into two independent $(\sigma \to \gamma)$-sequences f_1 and f_2 such that for all $a_1 \in [\![\varphi]\!]$ and $a_2 \in [\![\psi]\!]$, $[\![f]\!](a_1, a_2) = [\![f_1]\!](a_1) \cdot [\![f_2]\!](a_2)$. To do so, choose h_1 and h_2 such that $f = \lambda(x, y).h_1(x, y) \cdot h_2(x, y)$ (note that the total number of such choices is $|f| + 1$ where $|f|$ is the length of the output sequence), let $f_1 = \lambda x.h_1(x, \mathscr{W}(\psi))$, $f_2 = \lambda x.h_2(\mathscr{W}(\varphi), x)$ and check validity of the *split predicate*

$$\forall x\, y\, ((\varphi(x) \wedge \psi(y)) \Rightarrow f(x, y) = f_1(x) \cdot f_2(y))$$

If there exists a valid split predicate then pick such f_1 and f_2, and replace the above path with

$$(p, q) \xrightarrow{\varphi/(f_1,g)} (p'_1, q'_1) \xrightarrow{\psi/(f_2,g_1)} (p_2, q_2)$$

where (p'_1, q'_1) is a new *aligned* pair-state added to the frontier.
Suppose that splitting fails. We show that $A \not\stackrel{1}{=} B$, by way of contradiction. Assume $A \stackrel{1}{=} B$.
Since splitting fails, the following *dependency* predicates are satisfiable:

$$D1 = \lambda(x, x', y).\varphi(x) \wedge \varphi(x') \wedge \psi(y) \wedge f(x, y) \neq f(x', y)$$
$$D2 = \lambda(x, y, y').\varphi(x) \wedge \psi(y) \wedge \psi(y') \wedge f(x, y) \neq f(x, y')$$

Let $(a_1, a'_1, a_2) = \mathscr{W}(D1)$ and $(e_1, e_2, e'_2) = \mathscr{W}(D2)$. Assume that $A \stackrel{1}{=} B$. We proceed by case analysis over $|f|$. We know that $|f| \geq 1$, or else splitting is trivial.

(a) Assume first that $|f| = 1$. Let

$$[b] = [\![f]\!](a_1, a_2),\ [b'] = [\![f]\!](a'_1, a_2),\ [d] = [\![f]\!](e_1, e_2),\ [d'] = [\![f]\!](e_1, e'_2).$$

Thus $b \neq b'$ and $d \neq d'$. Since (p, q) is aligned, and (p_1, q_1) is reachable and alive (by construction of $A \times B$, $\bullet$ is reachable from (p_1, q_1)), there exists $\alpha, \beta \in \Sigma^*$, $u_1, u_2, v_1, v_2, v_3, v_4 \in \Gamma^*$, such that, by $IsSat(D1)$,

$$\left.\begin{array}{l} p_0 \xrightarrow{\alpha/u_1} p \xrightarrow{[a_1,a_2]/[b]} p_2 \xrightarrow{\beta/u_2}_A \bullet \\ q_0 \xrightarrow{\alpha/v_1} q \xrightarrow{[a_1]/[\![g]\!](a_1)} q_1 \xrightarrow{[a_2]\cdot\beta/v_2}_B \bullet \end{array}\right\} \stackrel{(A \stackrel{1}{=} B)}{\Longrightarrow} u_1 \cdot [b] \cdot u_2 = v_1 \cdot [\![g]\!](a_1) \cdot v_2$$

$$\left.\begin{array}{l} p_0 \xrightarrow{\alpha/u_1} p \xrightarrow{[a'_1,a_2]/[b']} p_2 \xrightarrow{\beta/u_2}_A \bullet \\ q_0 \xrightarrow{\alpha/v_1} q \xrightarrow{[a'_1]/[\![g]\!](a'_1)} q_1 \xrightarrow{[a_2]\cdot\beta/v_2}_B \bullet \end{array}\right\} \stackrel{(A \stackrel{1}{=} B)}{\Longrightarrow} u_1 \cdot [b'] \cdot u_2 = v_1 \cdot [\![g]\!](a'_1) \cdot v_2$$

By $b \neq b'$, $|v_1| \leq |u_1| < |v_1 \cdot [\![g]\!](a_1)| = |v_1| + |g|$. Also, by $IsSat(D2)$,

$$\left.\begin{array}{l} p_0 \xrightarrow{\alpha/u_1} p \xrightarrow{[e_1,e_2]/[d]} p_2 \xrightarrow{\beta/u_2}_A \bullet \\ q_0 \xrightarrow{\alpha/v_1} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \xrightarrow{[e_2]\cdot\beta/v_3}_B \bullet \end{array}\right\} \begin{array}{l}(A \overset{1}{=} B)\\ \Longrightarrow\end{array} \begin{array}{l} u_1 \cdot [d] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_3 \end{array}$$

$$\left.\begin{array}{l} p_0 \xrightarrow{\alpha/u_1} p \xrightarrow{[e_1,e'_2]/[d']} p_2 \xrightarrow{\beta/u_2}_A \bullet \\ q_0 \xrightarrow{\alpha/v_1} q \xrightarrow{[e_1]/[\![g]\!](e_1)} q_1 \xrightarrow{[e'_2]\cdot\beta/v_4}_B \bullet \end{array}\right\} \begin{array}{l}(A \overset{1}{=} B)\\ \Longrightarrow\end{array} \begin{array}{l} u_1 \cdot [d'] \cdot u_2 = \\ v_1 \cdot [\![g]\!](e_1) \cdot v_4 \end{array}$$

By $d \neq d'$, $|v_1 \cdot [\![g]\!](e_1)| = |v_1| + |g| \leq |u_1|$. But $|u_1| < |v_1| + |g|$. Ψ

(b) The case where $|f| > 2$ is similar to (a).

The remaining cases are similar and effectively eliminate all non-aligned pair-states from $A \times B$ or else establish that $A \not\overset{1}{=} B$. ⊠

Assume $A \times B$ is aligned and let $\lceil A \times B \rceil$ be the following *product SFT* (product ESFT all of whose transitions have lookahead 1) over the input type σ^*. For each $p \xrightarrow[\ell]{\lambda \bar{x}.\varphi(x_0,x_1,\ldots,x_{\ell-1})/(f,g)} q$ in $\Delta_{A\times B}$ let y be a variable of sort σ^* and let φ_1 be the σ^*-predicate

$$\lambda y.\varphi(y[0], y[1], \ldots, y[\ell-1]) \wedge tail^{\ell}(y) = [] \bigwedge_{i<\ell} tail^{i}(y) \neq []$$

where $y[i]$ is the term that accesses the i'th head of y and $tail^i(y)$ is the term that accesses the i'th tail of y. Lift f to the $(\sigma^* \to \gamma)$-sequence $f_1 = \lambda y.f(y[0], y[1], \ldots, y[\ell-1])$ and lift g similarly to g_1. Add the rule $p \xrightarrow[1]{\varphi_1/(f_1,g_1)} q$ as a rule of $\lceil A \times B \rceil$. Thus, the domain type of $\mathscr{T}_{\lceil A\times B\rceil}$ is $(\Sigma^*)^*$ while the range type is $2^{\Gamma^* \times \Gamma^*}$. For $u = [u_0, u_1, \ldots, u_n] \in (\Sigma^*)^*$, let $\lfloor u \rfloor \overset{\text{def}}{=} u_0 \cdot u_1 \cdots u_n$ in Σ^*.

Lemma 3 (Grouping). *Assume $A \times B$ is aligned. For all $u \in \Sigma^*$ and $v, w \in \Gamma^*$: $(v, w) \in \mathscr{T}_{A\times B}(u)$ iff $\exists z(u = \lfloor z \rfloor \wedge (v, w) \in \mathscr{T}_{\lceil A\times B\rceil}(z))$.*

Proof. The type lifting does not affect the semantics of the label-theory specific transformations. ⊠

Note that, $[[a_1, a_2], [a_3]]$ and $[[a_1], [a_2, a_3]]$ may be distinct inputs of the lifted product, while both correspond to the same flattened input $[a_1, a_2, a_3]$ of the original product. Intuitively, the internal subsequences correspond to input alignment boundaries of the two ESFTs A and B.

So, in particular, grouping preserves the property: there exists an input u and outputs $v \neq w$ such that $(v, w) \in \mathscr{T}_{A\times B}(u)$. We use the following lemma that is extracted from the main result in [16, Proof of Theorem 1].

Lemma 4 (SFT One-Equality [16]). *Let C be a product SFT over a decidable label theory. The problem of deciding if there exist u and $v \neq w$ such that $(v, w) \in \mathscr{T}_C(u)$ is decidable.*

We can now prove the main decidability result of this paper.

Theorem 6 (Cartesian ESFT One-Equality). *One-equality of Cartesian ESFTs over decidable label theories is decidable.*

Proof. Let A and B be Cartesian ESFTs. Construct $A \times B$. By the Product lemma 1, $\mathscr{D}(A \times B) = \mathscr{D}(A) \cap \mathscr{D}(B)$ and $A \not\stackrel{1}{=} B$ iff there exist u and $v \neq w$ such that $(v, w) \in \mathscr{T}_{A \times B}(u)$. By using the Alignment lemma 2, construct aligned product SFT C such that $\mathscr{T}_C = \mathscr{T}_{A \times B}$ or else determine that $A \not\stackrel{1}{=} B$. Now lift C to $\lceil C \rceil$, and by using the Grouping lemma 3, $A \not\stackrel{1}{=} B$ iff there exist u and $v \neq w$ such that $(v, w) \in \mathscr{T}_{\lceil C \rceil}(u)$. Finally, observe that adding the sequence operations for accessing the head and the tail of sequences in the lifting construction do, by themselves, not affect decidability of the label theory, apply Lemma 4. ⊠

3.3 Composition of ESFTs

In this section we show some preliminary results (mainly negative) on ESFT composition. In particular, ESFTs and Cartesian ESFTs are not closed under composition.

Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{x} \subseteq X$, $\mathbf{f}(\mathbf{x}) \stackrel{\text{def}}{=} \bigcup_{x \in \mathbf{x}} \mathbf{f}(x)$. Given $\mathbf{f} : X \to 2^Y$ and $\mathbf{g} : Y \to 2^Z$, $\mathbf{f} \circ \mathbf{g}(x) \stackrel{\text{def}}{=} \mathbf{g}(\mathbf{f}(x))$. This definition follows the convention in [5], i.e., $\circ$ applies first $\mathbf{f}$, then $\mathbf{g}$, contrary to how $\circ$ is used for standard function composition. The intuition is that $\mathbf{f}$ corresponds to the relation $R_{\mathbf{f}} : X \times Y$, $R_{\mathbf{f}} \stackrel{\text{def}}{=} \{(x, y) \mid y \in \mathbf{f}(x)\}$, so that $\mathbf{f} \circ \mathbf{g}$ corresponds to the binary relation composition $R_{\mathbf{f}} \circ R_{\mathbf{g}} \stackrel{\text{def}}{=} \{(x, z) \mid \exists y (R_{\mathbf{f}}(x, y) \wedge R_{\mathbf{g}}(y, z))\}$.

Definition 10. *A class of transducer C is closed under composition iff for every $\mathscr{T}_1$ and $\mathscr{T}_2$ that are C-definable $\mathscr{T}_1 \circ \mathscr{T}_2$ is also C-definable.*

Theorem 7 (Composition). *The following statements are true: ESFTs are not closed under composition; There exists two Cartesian ESFTs which composition is not ESFT definable; Cartesian ESFTs are not closed under composition.*

We now show that in general the composition of two ESFTs cannot be effectively computed.

Theorem 8 (Undecidability of Composition Computation). *Given two ESFTs with lookahead 2 over quantifier free successor arithmetic and tuples whose composition f is ESFT definable, it is undecidable to compute the ESFT corresponding to f.*

4 Experiments and Applications

In this section we show how several practical applications can be modelled and verified using ESFAs and ESFTs. We first use ESFTs to prove the correctness of some real world string encoders and decoders. We then show how ESFAs and ESFTs can be useful in the context of deep packet inspection and network protocol transformations. Finally we propose ESFTs as a tool for the analysis of list manipulating programs. All our experiments are run using the tool Bek[2].

[2] `http://www.rise4fun.com/Bek`.

Analysis of String Encoders. A string encoder E transforms input strings in a given format A into output strings in a different format B. A decoder D inverts such transformation. The formats A and B usually use different alphabets (character sets). The first half of Table 1 shows examples of common string encoders/decoders and their respective lookahead sizes. $E \circ D$ ($D \circ E$) denotes the sequential compositions of the encoder with the decoder (decoder with the encoder). We compute such compositions using the semi-decision procedure of [4].

The correctness of UTF8 encoding was already investigated in [4] using a semi-decision procedure for one-equality. We use the algorithm proposed in Section 3.2 to confirm such result and we prove the correctness of three new encoders: BASE64, BASE32 and BASE16. The second half of Table 1 shows the running times of the analyses. The column $E \circ D \stackrel{1}{=} I$ ($D \circ E \stackrel{1}{=} I$) shows the cost of checking whether $E \circ D$ ($D \circ E$) is one-equal to the identity transducer I. Composition times (typically 1-2 ms) are included in the measurements.

Table 1. Analysed encoders (E) and decoders (D), their lookaheads, and analysis times

	Lookahead		Analysis (ms)	
	E	D	$E \circ D \stackrel{1}{=} I$	$D \circ E \stackrel{1}{=} I$
UTF8:	2	4	16	24
BASE64:	3	5	53	19
BASE32:	5	8	8	12
BASE16:	1	2	2	1

We want to stress that during our experiments we identified wrong implementations of the UTF8 encoder/decoder in which the algorithm of Section 3.2 correctly detected that one-equality fails, while the semi-decision procedure used in [4] did not terminate.

Deep Packet Inspection. Fast identification of network traffic patterns is of vital importance in network routing, firewall filtering and intrusion detection. This task is addressed with the name "deep packet inspection" (DPI) [15]. Due to performance constraints, DPI must be performed in a single pass over the input. The simplest approach is to use DFAs and NFAs to identify patterns. These representations are either not succinct or not streamable. Extended Finite Automata (XFA) [15] make use of registers to reduce the state space while preserving determinism and therefore deterministic ESFAs can be seen as a subclass of XFAs that are able to deal with finite lookahead. Deterministic ESFA can also represent the alphabet symbolically, which enables a new level of succinctness. We believe that deterministic ESFAs can help achieve further succinctness in particular problem instances. To support this hypothesis we observe that several examples shown in [15, Figure 2,3] can be represented as deterministic ESFAs with few transitions. For example the language `^/\ncmd[^\n]{200}$` can be succinctly captured by a deterministic ESFA with one transition!

Network Protocol Conversions. Deep packet inspection can be naturally extended by adding data manipulation. As in the previous setting we are interested in deterministic ESFTs which can commit their output at every transition without seeing the rest of the input. Deterministic ESFTs can be used to compute logs of network traffic or translate headers of one protocol into another. As an example, a simple translation from an IPv4 header to an IPv6 header[3] can

[3] More information at `http://www.cs.washington.edu/research/networking/napt/`

be easily implemented with a deterministic ESFT with less than 50 transitions. The same transformation using an SFT would require over 100000 transitions.

Verification of List Manipulating Programs. ESFTs can be used for verification of *list manipulating programs* as they naturally model sequential pattern matching. The ML guards `x1::x2::xs -> (x1+x2)::(f2 xs)` and `x1::x2::x3::xs -> (x1+x2+x3)::(f3 xs)`, respectively belonging to the functions $f_2, f_3 : list\ int \to list\ int$, can be naturally expressed as ESFT transitions. Therefore f_2 and f_3 can be modelled as ESFTs. We can then use the one-equality algorithm of Section 3.2 to prove that $f_2(f_2(f_2\ l)) \overset{1}{=} f_3(f_3\ l)$ in less than 1 ms.

5 Related Work

Symbolic finite transducers (SFTs) and Bek were originally introduced in [7] with a focus on security analysis of sanitizers. The formal foundations and the theoretical analysis of the underlying SFT algorithms, in particular, an algorithm for one-equality of SFTs, modulo a decidable background theory is studied in [16]. Symbolic Transducers (STs) that allow the use of registers are also defined in [16]. Full equivalence of finite state transducers is undecidable [6], and already so for very restricted fragments [8]. In the single-valued case, decidability was established in [13], and extended to the finite-valued case in [3,17].

ESFTs were introduced in [4] as a succinct and more analysable representation of a subclass of symbolic transducers (STs). The main result in [4] is a register elimination technique that provides a way to construct (product) ESFTs from (product) symbolic transducers (STs). While this technique provides a semi-decision procedure for one-equality checking (by using grouping, Lemma 3) of non-Cartesian ESFTs, it does not provide a full decision procedure for one-equality of the Cartesian case. The procedure in [4] fails to decide *alignment* of ESFTs, that is the key lemma (Lemma 2) used in the main decidability result of Theorem 6, that is a proper extension of the decidability result of one-equality of SFTs [16, Theorem 1]. We also show that one-equality is undecidable in the non-Cartesian case, Theorem 5, that is in sharp contrast to the theory of classical automata, where the non-Cartesian case is irrelevant (from the point of view of decidability) due to the standard form [18, Theorem 2.17].

Extended Top-Down Tree Transducers [11] (ETTTs) are commonly used in natural language processing. ETTTs also allow finite lookahead on transformation from trees to trees, but only support finite alphabets. The special case in which the input is a string (unary tree) is equivalent to ESFTs over finite alphabets. This paper focuses on ESFTs over any decidable theory. We leave as future work extending the model to tree transformations.

Symbolic finite transducers with lookback k (k-SLTs) [2] have a sliding window of size $k+1$ that allows, in addition to the current input character, references of up to k previous characters. SLTs use only final states, because it is unclear how to support nonfinal states in the context of learning. Thus, domain intersection (that is udecidable for ESFTs with lookahead 2) is trivial for SLTs. Another fundamental difference between ESFTs and SLTs is in their semantics:

all SLT transitions read one character at a time, while an ESFT transition with lookahead k reads k characters at once.

In recent years there has been considerable interest in automata using infinite alphabets [14], starting with the work on register automata [9]. Finite words over an infinite alphabet are often called data words. This line of work focuses on fundamental questions about decidability, complexity, and expressiveness on classes of automata on one hand and fragments of logic on the other hand.

Streaming transducers [1] provide another recent symbolic extension of finite transducers where the label theories are restricted to be total orders, in order to maintain decidability of equivalence. Streaming transducers are largely orthogonal to SFTs or the extension of ESFTs, as presented in the current paper. For example, streaming transducers do not allow arithmetic, but can reverse the input, which is not possible with ESFTs.

The correctness of Utf8 encoder and decoder was proven in [4] using two semi-decision procedures for equivalence and composition. In this paper we show that the composition of Utf8 encoder and decoder can be expressed as a Cartesian ESFT and can be formally analyzed with the one-equality algorithm introduced in this paper. We do the same for three encoders of which the correctness was not proven before: Base64, Base32, Base16.

Extended Finite Automata (XFA) are introduced in [15] for network packet inspection. XFAs are a succinct representation of DFAs that use registers and allow programs over the registers. ESFAs are orthogonal to XFAs in two ways: 1) XFAs only support finite alphabets; and 2) XFAs aim at representing *most* DFAs succinctly, while ESFAs only capture the languages that use finite lookahead. We have not investigated the application of ESFAs to network packet inspection in detail, but we think that they can help achieving a further level of succinctness. History-based finite automata [10] are another extension of DFAs that have been introduced for encoding regular expressions in the context of network intrusion detection systems, they use a single register (bit-vector) to keep track of history. The register is used together with the input character to determine when a transition is enabled.

6 Conclusion

We showed fundamental negative and positive results about several classical decision problems of ESFAs and ESFTs, establishing a sharp boundary between decidability (the Cartesian case with any decidable background) and undecidability (the non-Cartesian case with a background of successor arithmetic). While the main motivation came from typical *static analysis* problems using ESFTs, an equally important application of the Cartesian case is for efficient *code generation*. Namely, the conjuncts of a Cartesian predicate can be compiled and normalized into separate unary predicates that may for example use BDDs for efficient and unique set representation when dealing with bit-vectors, as in the context of strings coders. Identifying classes of ESFTs that are closed under composition, as well as extending ESFTs to trees are left as open problems.

References

1. Alur, R., Cerný, P.: Streaming transducers for algorithmic verification of single-pass list-processing programs. In: POPL 2011, pp. 599–610. ACM (2011)
2. Botincan, M., Babic, D.: Sigma*: symbolic learning of input-output specifications. In: POPL 2013, pp. 443–456. ACM (2013)
3. Culic, K., Karhumäki, J.: The equivalence of finite-valued transducers (on HDTOL languages) is decidable. Theoretical Computer Science 47, 71–84 (1986)
4. D'Antoni, L., Veanes, M.: Static analysis of string encoders and decoders. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 209–228. Springer, Heidelberg (2013)
5. Fülöp, Z., Vogler, H.: Syntax-Directed Semantics: Formal Models Based on Tree Transducers. EATCS (1998)
6. Griffiths, T.: The unsolvability of the equivalence problem for Λ-free nondeterministic generalized machines. J. ACM 15, 409–413 (1968)
7. Hooimeijer, P., Livshits, B., Molnar, D., Saxena, P., Veanes, M.: Fast and precise sanitizer analysis with Bek. In: USENIX Security, pp. 1–16 (2011)
8. Ibarra, O.: The unsolvability of the equivalence problem for Efree NGSM's with unary input (output) alphabet and applications. SIAM Journal on Computing 4, 524–532 (1978)
9. Kaminski, M., Francez, N.: Finite-memory automata. TCS 134(2), 329–363 (1994)
10. Kumar, S., Chandrasekaran, B., Turner, J., Varghese, G.: Curing regular expressions matching algorithms from insomnia, amnesia, and acalculia. In: ANCS 2007, pp. 155–164. ACM/IEEE (2007)
11. Maletti, A., Graehl, J., Hopkins, M., Knight, K.: The power of extended top-down tree transducers. SIAM J. Comput. 39(2), 410–430 (2009)
12. Mohri, M.: Finite-state transducers in language and speech processing. Comput. Linguist. 23(2), 269–311 (1997)
13. Schützenberger, M.P.: Sur les relations rationnelles. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 209–213. Springer, Heidelberg (1975)
14. Segoufin, L.: Automata and logics for words and trees over an infinite alphabet. In: CSL, pp. 41–57 (2006)
15. Smith, R., Estan, C., Jha, S., Kong, S.: Deflating the big bang: fast and scalable deep packet inspection with extended finite automata. In: SIGCOMM 2008, pp. 207–218. ACM (2008)
16. Veanes, M., Hooimeijer, P., Livshits, B., Molnar, D., Bjorner, N.: Symbolic finite state transducers: Algorithms and applications. In: POPL 2012, pp. 137–150. ACM (2012)
17. Weber, A.: Decomposing finite-valued transducers and deciding their equivalence. SIAM Journal on Computing 22(1), 175–202 (1993)
18. Yu, S.: Regular languages. In: Rozenberg, G., Salomaa, A. (eds.) Handbook of Formal Languages, vol. 1, pp. 41–110. Springer (1997)

Finite Model Finding in SMT*

Andrew Reynolds[1], Cesare Tinelli[1], Amit Goel[2], and Sava Krstić[2]

[1] Department of Computer Science, The University of Iowa
[2] Strategic CAD Labs, Intel Corporation

Abstract. SMT solvers have been used successfully as reasoning engines for automated verification. Current techniques for dealing with quantified formulas in SMT are generally incomplete, forcing SMT solvers to report "unknown" when they fail to prove the unsatisfiability of a formula with quantifiers. This inability to return counter-models limits their usefulness in applications that produce quantified verification conditions. We present a novel finite model finding method that reduces these limitations in the case of quantifiers ranging over free sorts. Our method contrasts with previous approaches for finite model finding in first-order logic by not relying on the introduction of *domain constants* for the free sorts and by being fully integrated into the general architecture used by most SMT solvers. This integration is achieved through the addition of a novel solver for sort cardinality constraints and a module for quantifier instantiation over finite domains. Initial experiments with verification conditions generated from a deductive verification tool developed at Intel Corp. show that our approach compares quite favorably with the state of the art in SMT.

1 Introduction

Techniques and solvers for Satisfiability Modulo Theories (SMT) have been used successfully in recent years to support a variety of formal methods for hardware and software development, including automated verification. They are especially effective for verification tasks that can be reduced to proving the unsatisfiability of quantifier-free formulas in certain logical theories. A number of verification applications, however, dealing with data structures not modeled by an SMT solver's built-in theories, or analyzing systems with an unbounded number of processes or memory locations, require solvers that can prove the unsatisfiability of *quantified* formulas in those theories.

SMT solvers that can reason about quantified formulas are based on incomplete methods and so often report "unknown" when they fail, after some predetermined amount of effort, to prove a quantified formula unsatisfiable. For verification purposes, however, it is very useful to know when such formulas are indeed satisfiable; especially if the solver can also return some representation of the formula's model, as that can be used to identify errors in the artifact being verified or in the formulation of its intended properties. Current SMT solvers are able to produce models of satisfiable quantified formula only in fairly restricted cases [9], which limits their scope and usefulness.

We reduce these limitations with a novel method for model finding in SMT. By the undecidability of first-order logic there are no automated methods for finding arbitrary

* The work of the first two authors was partially funded by a grant from Intel Corporation.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 640–655, 2013.

models. So we focus on *finite* models, which can be enumerated and represented symbolically. More precisely, since SMT solvers work with sorted logics with both built-in and *free* ("uninterpreted") sorts, we focus on finding models that interpret the free sorts as finite domains. As with finite model finders for standard first-order logic, the main idea is simply to check universal quantifiers exhaustively over candidate models with increasingly large domains for the free sorts, until an actual model is found. Our method differs from previous approaches by not relying on the explicit introduction of *domain constants* for the free sorts, and by being fully integrated into the general architecture used by many SMT solvers. While our approach is limited to SMT formulas with quantifiers ranging only over free sorts, it is still quite useful because such formulas occur often in verification applications; moreover, when satisfiable they usually have small finite models.

We present our finite model finding method in the context of an abstract framework that models a large class of SMT solvers supporting multiple theories and quantified formulas. An overview of this framework is provided in Section 2. The method itself is described in Section 3. In Section 4, we discuss the initial experimental results obtained with our implementation of the method within the SMT solver CVC4.

Related Work. The state of the art in finite model finding in first-order logic is exemplified by tools such as MACE and Paradox [6]. Their approach is based on encoding to SAT the problem of whether a given set of universally quantified first-order formulas has a model of a given size k. The encoding is based on (i) the introduction of k *domain constants*, fresh constant symbols representing the elements of the model's domain; and (ii) an exhaustive instantiation of the input formulas with these constants. Further ground constraints are added to state that the k domain constants denote all domain elements and are pairwise distinct. The resulting ground formulas are then translated to an equisatisfiable propositional formula and fed to a SAT solver. Advances on this approach mostly focus on addressing two of its main limitations for scalability: the size of the resulting propositional formula and the presence of *value symmetries*, that is, partial solutions that are equivalent modulo a permutation of the domain constants (see [16] for example). One way to drastically reduce the size of the final formula is to encode the problem in a more expressive, but still decidable, fragment first-order logic such as, for instance, function-free clause logic [4].

A completely different approach, pioneered by the SEM model finder [18], is based on traditional constraint satisfaction methods and a built-in treatment of ground equational reasoning. It relies on special search techniques, such as symmetry reduction and least-number heuristics, to enumerate possible models efficiently.

Our method is more similar to SEM-style model finding in that it is not based on reductions to SAT, and is free of the spurious symmetries caused by the use of domain constants. In contrast to SEM, we rely on the DPLL(T) architecture common to many SMT solvers as our core search procedure, and use on-demand instantiation techniques for handling quantifiers. That way, our method can handle natively formulas that also involve operators from theories such as arithmetic, arrays, bit vectors, and so on, as long as none of their quantifiers range over the non-free sorts of these theories.

Formal Preliminaries. We work in the context of many-sorted first-order logic with equality. We fix a set $\mathbf{S}$ of *sort symbols* and for every $S \in \mathbf{S}$ an infinite set of $\mathbf{X}_S$ of

variables of sort S. We assume the sets $\mathbf{X}_S$ are pairwise disjoint and let $\mathbf{X}$ be their union. A *signature* Σ consists of a set $\Sigma^{\mathrm{s}} \subseteq \mathbf{S}$ of sort symbols and a set Σ^{f} of *(sorted) function symbols* $f^{S_1 \cdots S_n S}$, where $n \geq 0$ and $S_1, \ldots, S_n, S \in \Sigma^{\mathrm{s}}$. We drop the sort superscript from function symbols when it is clear from context or unimportant. Without loss of generality we use equality, denoted by $\approx$, as the only predicate symbol. We abbreviate $\neg(s \approx t)$ with $s \not\approx t$.

Given a signature Σ, well-sorted terms, atoms, literals, clauses, and (possibly quantified) formulas with variables in $\mathbf{X}$ are defined as usual[1] and referred to respectively as Σ-*terms*, Σ-*atoms* and so on. A *ground term/formula* is a Σ-term/formula with no variables. Where $\mathbf{x} = (x_1, \ldots, x_n)$ is tuple of variables and Q is either $\forall$ or $\exists$, we write $Q\mathbf{x}\,\varphi$ as an abbreviation of $Qx_1 \cdots Qx_n\,\varphi$. If φ is a Σ-formula and $\mathbf{x}$ has no repeated variables, we write $\varphi[\mathbf{x}]$ to denote that φ's free variables are from $\mathbf{x}$; if $\mathbf{t} = (t_1, \ldots, t_n)$ is a term tuple we write $\varphi[\mathbf{t}]$ for the formula obtained from φ by simultaneously replacing each occurrence of x_i in φ by t_i.

A Σ-*interpretation* I maps: each $S \in \Sigma^{\mathrm{s}}$ to a non-empty set S^I, the *domain* of S in I; each $x \in \mathbf{X}$ of sort S to an element $x^I \in I_S$; and each $f^{S_1 \cdots S_n S} \in \Sigma^{\mathrm{f}}$ to a total function $f^I : S_1^I \times \cdots \times S_n^I \to S^I$. A satisfiability relation between Σ-interpretations and Σ-formulas is defined inductively as usual. The *reduct* of I to a sub-signature Ω of Σ is an Ω-interpretation that coincides with I on the symbols in Ω.

A *theory* is a pair $T = (\Sigma, \mathbf{I})$ where Σ is a signature and $\mathbf{I}$ a class of Σ-interpretations, the *models* of T, that is closed under variable reassignment (i.e., every Σ-interpretation that differs from one in $\mathbf{I}$ only for how it interprets the variables is also in $\mathbf{I}$) and isomorphism. A formula $\varphi[\mathbf{x}]$ of T is *satisfiable* (resp., *unsatisfiable*) *in* T if it is satisfied by some (resp., no) interpretation in $\mathbf{I}$. A set Γ of formulas *entails in* T a Σ-formula φ, written $\Gamma \models_T \varphi$, if every interpretation in $\mathbf{I}$ that satisfies all formulas in Γ satisfies φ as well. The set Γ is *satisfiable in* T if $\Gamma \not\models_T \bot$ where $\bot$ is the universally false atom. When Γ and φ are ground we write $\Gamma \models_{\mathrm{p}} \varphi$ if Γ *propositionally entails* φ, that is, if the set $\Gamma \cup \{\neg\varphi\}$ is unsatisfiable when considering all atomic formulas in it as propositional variables. The *combination* $T_1 + T_2$ of two theories $T_1 = (\Sigma_1, \mathbf{I}_1)$ and $T_2 = (\Sigma_2, \mathbf{I}_2)$ is the theory $(\Sigma, \mathbf{I})$ where $\Sigma = \Sigma_1 \cup \Sigma_2$ and $\mathbf{I}$ is the largest class of Σ-interpretations whose reduct to Σ_i is in $\mathbf{I}_i$ for $i = 1, 2$.

2 Satisfiability Modulo Multiple Theories

In its most general formulation, SMT is the problem of determining the satisfiability of a set of formulas in some theory T which is possibly a combination of several theories. Our finite model finding method applies to *lazy* SMT solvers based on the DPLL(T) architecture [12]. Such solvers combine modularly a generic CDCL SAT solver[2] (the *SAT engine*) with one or more reasoners (the *theory solvers*) specialized on deciding the satisfiability of *constraints*, conjunctions of ground literals, in a specific theory. Some SMT solvers are able to reason also about quantified formulas. All of them rely on some form of *heuristic quantifier instantiation* where existential quantifiers are Skolemized and universal ones are instantiated with a heuristic selection of ground terms.

[1] With atoms $s \approx t$ well sorted iff s and t are well sorted terms of the *same* sort.

[2] Conflict-Driven Clause Learning solvers were previously referred to as *DPLL* solvers.

DPLL(T)-style SMT solvers are conveniently described at an abstract level using a rule-based framework introduced by Nieuwenhuis *et al.* [12] and then further developed by Krstić and Goel [11] for solvers that combine multiple built-in theories, and by Ge *et al.* [8] for solvers that use heuristic quantifier instantiation. We synthesize the main ideas of those works in a single framework, focusing on aspects most relevant to our task at hand. We present the general framework here and then show in Section 3 how it can be extended to look for finite models for formulas with quantifiers over free sorts.

Abstract Framework. For the rest of the paper we will consider a theory $T = T_1 + \ldots + T_m$ where each T_i is a theory of signature Σ_i, and T_1 is the theory of equality over "uninterpreted functions", also known as EUF. We call *free* those sort and function symbols whose interpretation is not restricted in any way by any of the theories, and consider them as part of the EUF signature; we call *built-in* all the others. For convenience and without loss of generality, we assume that $\Sigma_1, \ldots, \Sigma_m$ have the same set $\mathbf{S}$ of sort symbols and share a distinguished infinite set $\mathcal{C}_S$ of free constants of sort S for each $S \in \mathbf{S}$. Let $\mathcal{C} = \bigcup_{S \in \mathbf{S}} \mathcal{C}_S$. We also assume that $\mathbf{S}$ includes a Boolean sort Bool and a constant true of that sort—allowing us to encode predicate symbols as function symbols of return sort Bool. As customary, we impose the (real) restriction that the signatures $\Sigma_1, \ldots, \Sigma_m$ share no function symbols besides the constants in $\mathcal{C}$.

We describe SMT solvers for the theory T abstractly as state transition systems. States are either the distinguished state fail or triples of the form $\langle M, F, C \rangle$ where

- M, the current *assignment*, is a sequence of literals and *decision points* $\bullet$,
- F is a set of formulas derived from the original input problem, and
- C is either the distinguished value no or a *conflict clause*.

Each assignment M can be factored uniquely into the subsequence concatenation $M_0 \bullet M_1 \bullet \cdots \bullet M_n$, where no M_i contains decision points. For $i = 0, \ldots, n$, we call M_i the *decision level* i of M and denote with $M^{[i]}$ the subsequence $M_0 \bullet \cdots \bullet M_i$. When convenient, we will treat M as the set of its literals and call them the *asserted literals*.

The formulas in F have a particular *purified form* that can be assumed with no loss of generality since any formula can be efficiently converted into that form while preserving satisfiability in T: each element of F is either a ground clause or a formula of the form $a \Leftrightarrow \forall \mathbf{x} C[\mathbf{x}]$ where a is a ground atom and C is a clause. Moreover, each atom occurring in F is *pure*, that is, has signature Σ_i for some $i \in \{1, \ldots, m\}$.

Initial states have the form $\langle \emptyset, F_0, \mathsf{no} \rangle$ where F_0 is an input set of formulas to be checked for satisfiability. The expected final states are fail, when F_0 is unsatisfiable in T; or $\langle M, F, \mathsf{no} \rangle$ with M satisfiable in T, F equisatifiable with F_0 in T, and $M \models_{\mathrm{p}} F$.

Transition Rules. The possible behaviors of the system are defined by a set of non-deterministic state transition rules, specifying a set of successor states for each current state.[3] The rules are provided in Figure 1 in *guarded assignment form* [11]. A rule applies to a state s if all of its premises hold for s. In the rules, M, F and C respectively denote the assignment, formula set, and conflict component of the current state. The conclusion describes how each component is changed, if at all. We write $\overline{l}$ to denote the

[3] To simplify the presentation, we do not consider here rules that model the forgetting of learned lemmas and restarts of the SMT solver.

$$\textbf{Decide}\ \frac{l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l, \overline{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M} \bullet l} \qquad \textbf{Conflict}_i\ \frac{\mathsf{C} = \mathsf{no} \quad l_1, \ldots, l_n \in \mathsf{M} \quad l_1, \ldots, l_n \models_i \bot}{\mathsf{C} := \overline{l}_1 \vee \cdots \vee \overline{l}_n}$$

$$\textbf{Fail}\ \frac{\mathsf{C} \neq \mathsf{no} \quad \bullet \notin \mathsf{M}}{\mathsf{fail}} \qquad \textbf{Backjump}\ \frac{\mathsf{C} = l_1 \vee \cdots \vee l_n \vee l \quad \mathsf{lev}\ \overline{l}_1, \ldots, \mathsf{lev}\ \overline{l}_n \leq\ i < \mathsf{lev}\ \overline{l}}{\mathsf{C} := \mathsf{no} \quad \mathsf{M} := \mathsf{M}^{[i]}\ l}$$

$$\textbf{Propagate}_i\ \frac{l_1, \ldots, l_n \in \mathsf{M} \quad l_1, \ldots, l_n \models_i l \quad l \in \mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}} \quad l, \overline{l} \notin \mathsf{M}}{\mathsf{M} := \mathsf{M}\ l}$$

$$\textbf{Explain}_i\ \frac{\mathsf{C} = l \vee D \quad \overline{l}_1, \ldots, \overline{l}_n \models_i \overline{l} \quad \overline{l}_1, \ldots, \overline{l}_n \prec_{\mathsf{M}} \overline{l}}{\mathsf{C} := l_1 \vee \cdots \vee l_n \vee D} \qquad \textbf{Learn}\ \frac{\mathsf{C} \neq \mathsf{no}}{\mathsf{F} := \mathsf{F} \cup \{\mathsf{C}\}}$$

$$\textbf{Learn}_i\ \frac{\emptyset \models_i \exists \mathbf{x}\,(l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}]) \quad l_1, \ldots, l_n \in \mathrm{Lit}_{\mathsf{M}}|_i \cup \mathrm{Int}_{\mathsf{M}} \cup L_i}{\mathsf{F} := \mathsf{F} \cup \{l_1[\mathbf{c}] \vee \cdots \vee l_n[\mathbf{c}]\}}$$

$$\forall\textbf{-Inst}\ \frac{a \in \mathsf{M} \quad a \Leftrightarrow \forall \mathbf{x} C[\mathbf{x}] \in \mathsf{F}}{\mathsf{F} := \mathsf{F} \cup \{\neg a \vee C[\mathbf{t}]\}} \qquad \exists\textbf{-Inst}\ \frac{\neg a \in \mathsf{M} \quad a \Leftrightarrow \forall \mathbf{x} l_1[\mathbf{x}] \vee \cdots \vee l_n[\mathbf{x}] \in \mathsf{F}}{\mathsf{F} := \mathsf{F} \cup \{a \vee \overline{l}_1[\mathbf{c}], \ldots, a \vee \overline{l}_n[\mathbf{c}]\}}$$

Fig. 1. DPLL($T_1, \ldots, T_m$) rules. In **Learn**$_i$, $\mathbf{x}$ may be empty. In **Learn**$_i$ and $\exists$**-Inst**, $\mathbf{c}$ are fresh constants from $\mathcal{C}$ of the same sort as $\mathbf{x}$. In $\forall$**-Inst**, $\mathbf{t}$ are ground terms of the same sort as $\mathbf{x}$ and such that $C[\mathbf{t}]$ is in purified form.

complement of literal l and $l \prec_{\mathsf{M}} l'$ to indicate that l occurs before l' in M. The function lev maps each literal of M to the (unique) decision level at which l occurs in M. The set $\mathrm{Lit}_{\mathsf{F}}$ (resp., $\mathrm{Lit}_{\mathsf{M}}$) consists of all ground literals in F (resp., all literals of M) and their complements. For $i = 1, \ldots, m$, the set $\mathrm{Lit}_{\mathsf{M}}|_i$ consists of the Σ_i-literals of $\mathrm{Lit}_{\mathsf{M}}$. $\mathrm{Int}_{\mathsf{M}}$ is the set of all *interface literals* of M: the equalities and disequalities between constants c, d with c and d occurring in $\mathrm{Lit}_{\mathsf{M}}|_i$ and $\mathrm{Lit}_{\mathsf{M}}|_j$ for two distinct $i, j \in \{1, \ldots, m\}$.

The index i ranges from 0 to m for the rules **Propagate**$_i$, **Conflict**$_i$ and **Explain**$_i$, and from 1 to m for **Learn**$_i$. In all rules, $\models_i$ abbreviates $\models_{T_i}$ when $i > 0$. In **Propagate**$_0$, $l_1, \ldots, l_n \models_0 l$ simply means that $\overline{l}_1 \vee \cdots \vee \overline{l}_n \vee l \in \mathsf{F}$. Similarly, in **Conflict**$_0$, $l_1, \ldots, l_n \models_0 \bot$ means that $\overline{l}_1 \vee \cdots \vee \overline{l}_n \in \mathsf{F}$; in **Explain**$_0$, $\overline{l}_1, \ldots, \overline{l}_n \models_0 \overline{l}$ means that $l_1 \vee \cdots \vee l_n \vee \overline{l} \in \mathsf{F}$. The rules **Decide**, **Propagate**$_0$, **Explain**$_0$, **Conflict**$_0$, **Fail**, **Learn**, and **Backjump** model the behavior of the SAT engine, which treats ground atoms as Boolean variables and ignores quantified formulas. The rules **Conflict**$_0$ and **Explain**$_0$ model the conflict discovery and analysis mechanism used by CDCL SAT solvers.

All the other rules but $\forall$**-Inst** and $\exists$**-Inst** model the interaction between the SAT engine and the individual theory solvers in the overall SMT solver. Generally speaking, the system uses the SAT engine to construct the assignment M as if the problem were propositional, but it periodically asks the sub-solvers for each theory T_i to check if the set of Σ_i-constraints in M is unsatisfiable in T_i, or entails some yet undetermined literal from $\mathrm{Lit}_{\mathsf{F}} \cup \mathrm{Int}_{\mathsf{M}}$. In the first case, the sub-solver returns an *explanation* of the unsatisfiability as a conflict clause, which is modeled by **Conflict**$_i$ with $i = 1, \ldots, m$. The propagation of entailed theory literals and the extension of the conflict analysis mechanism to them is modeled by the rules **Propagate**$_i$ and **Explain**$_i$. The inclusion of the interface literals $\mathrm{Int}_{\mathsf{M}}$ in **Decide** and **Propagate**$_i$ achieves the effect of the

Nelson-Oppen combination method [15, 5]. The rule **Learn**$_i$ is needed to model theory solvers following the splitting-on-demand paradigm [3]. When asked about the satisfiability of their constraints, these solvers, may return instead a *splitting lemma*, a formula valid in their theory and encoding a guess that needs to be made about the constraints before the solver can determine their satisfiability. The set L_i in the rule is a finite set consisting of literals, not present in the original formula F_0, which may be generated by such solvers.

The **∀-Inst** and **∃-Inst** rules model the quantifier instantiation mechanism. When the atom a, which serves as a proxy for the quantified formula $\forall \mathbf{x} C$, occurs positively in the current assignment M, the SMT solver adds one or more ground instances of the clause $a \Rightarrow C[\mathbf{x}]$. When a occurs negatively, the system adds the (Skolemized) clause form of $\neg a \Rightarrow \neg\forall \mathbf{x} C$. Instantiation heuristics dictate which instances are generated and how quantifier instantiation applications are interleaved with the other operations.

Executions and Correctness. An *execution* of a transition system modeled as above is a (possibly infinite) sequence $s_0, s_1, \ldots$ of states such that s_0 is an initial state and for all $i \geq 0$, s_{i+1} can be generated from s_i by the application of one of the transition rules. A system state is *irreducible* if no transition rules besides **Learn**$_i$ apply to it. An *exhausted execution* is a finite execution whose last state is irreducible. An application of **Learn**$_i$ is *redundant* in an execution if the execution contains a previous application of **Learn**$_i$ with the same premise.

Adapting results from [12, 11, 3], it can be shown that every execution satisfies the following invariants: M contains only pure literals and no repetitions; $\mathsf{F} \models_T \mathsf{C}$ and $\mathsf{M} \models_{\mathrm{p}} \neg\mathsf{C}$ when $\mathsf{C} \neq \mathsf{no}$; every model of T satisfying F satisfies the initial set of formulas. Moreover, in the absence of quantified formulas, the transition system is *terminating*: every execution with no redundant applications of **Learn**$_i$ is finite; and *sound*: for every execution starting with a state $\langle \emptyset, F_0, \mathsf{no} \rangle$ and ending with fail, the clause set F_0 is unsatisfiable in T. Under suitable assumptions on the sub-theories $T_1, \ldots, T_m$, the system is also *complete*: for every exhausted execution starting with $\langle \emptyset, F_0, \mathsf{no} \rangle$ and ending with $\langle M, F, \mathsf{no} \rangle$, M is satisfiable in T and $M \models_{\mathrm{p}} F_0$. With quantified formulas, soundness is preserved but termination and completeness are lost in general.

3 Finite Model Finding with DPLL$(T_1, \ldots, T_m)$

We have developed a method that, given a set F_0 of formulas in purified form, searches for a model of T that satisfies F_0 and interprets all the free sorts as finite sets. Abusing the terminology, we will call such models *finite*.

The basic version of our method is restricted to input sets whose quantified formulas quantify only variables of free sorts. An extended version applies also to quantifiers over built-in sorts such as integer, real, array sorts and so on, as long as the quantified variables occur only as arguments of free function symbols. However, we do not discuss that extension here for space constraints.[4] In the basic version, thanks to the

[4] In fact, the extended version also works with quantifiers over built-in sorts always interpreted as a fixed finite domain such as, for instance, the sorts in the theory of fixed sized bit vectors. However, it is practical only for domains of small size.

use of formulas in purified form, our treatment of terms constructed with built-in function symbols is completely standard: built-in ground literals are processed modularly by their corresponding theory solver, and global consistency of the asserted literals is guaranteed via the exchange of interface literals. As a consequence, we focus on free function symbols here.

We look for finite models with the aid of a new theory FCC (finite cardinality constraints) and a solver for it. We assume FCC is one of the sub-theories $T_1, \ldots, T_m$.

Definition 1 (Theory FCC **of finite cardinality constraints).** *The signature* Σ_{FCC} *of* FCC *consists of (i) the same free sort symbols of* EUF*, (ii) the set* $\mathcal{C}$ *of free constants, and (iii) a constant* $\mathsf{card}_{S,k}$ *of sort* Bool *for each free sort* S *and integer* $k > 0$*. Its models are all* Σ_{FCC}*-interpretations that satisfy each* $\mathsf{card}_{S,k}$ *exactly when they interpret* S *as a set of cardinality* $n \leq k$*.*

Note that the only ground atoms in FCC besides those of the form $\mathsf{card}_{S,k}$ are equalities between free constants. It is not difficult to show, using reductions to and from graph coloring, that the satisfiability of ground literals in FCC is an NP-complete problem. A solver for ground EUF constraints and one for ground FCC constraints can be combined Nelson-Oppen style to obtain a solver for *ground* EUF *problems with finite cardinality constraints*. This follows immediately from extended combination results by Ranise *et al.* ([13], Theorems 13 and 21). The main idea is to apply the standard Nelson-Oppen non-deterministic combination procedure [15] but to a *flattened* version of the the original input problem, a set of equational literals with equations and disequation respectively of the form $c \approx f(c_1, \ldots, c_n)$ and $d \not\approx d'$ where f is a symbol of the original problem and $c, c_1, \ldots, c_n, d, d'$ are constants from $\mathcal{C}$. We will call this form a *flat form*. This entails that the theory FCC can be incorporated into our abstract framework without loss of completeness for ground problems, provided that all literals in the problem are in flat form.[5]

3.1 An Efficient Solver for FCC

We have developed an FCC solver meant to be efficient in practice when integrated into the DPLL($T_1, \ldots, T_m$) architecture together with a conventional congruence closure-based solver for EUF. We describe how to use the FCC solver to endow DPLL($T_1, \ldots, T_m$) with finite model finding abilities for EUF in the next subsection. Here, we give a high level overview of the FCC solver and how it cooperates with the EUF solver to solve ground EUF problems with cardinality constraints.

The main idea is first to find a model of the EUF constraints, if it exists; and then try to *shrink* that model as needed to satisfy the cardinality constraints. Consider the constraints $G \cup R$ where

- G is a set of equalities and disequalities in flat form
- R is a set of FCC constraints over the free sorts of EUF
- *any (dis)equality between free constants that occurs in* R *is also in* G

[5] In reality, a flat form is not needed. One can construct an FCC solver that takes in arbitrary ground EUF literals but treats every EUF term as a constant.

Let $\mathbf{T}_G$ be the set of all (sub-)terms occurring in G. If G is satisfiable, the EUF solver can compute a congruence relation $\equiv_E$ over $\mathbf{T}_G$ that is consistent with G in the sense that $s \equiv_E t$ for all $s \approx t \in E$ and $c \not\equiv_E d$ for all $c \not\approx d \in D$. Since G is in flat form, each equivalence class of $\equiv_E$ of terms of some sort S contains a constant from $\mathcal{C}_S$, so we can use it as the representative of that class and call it a *representative constant for S*. It is well-known that if $c_1, \ldots, c_h$ are all the representative constants for a free sort S, there is an EUF model $\mathcal{M}$ satisfying G such that $S^{\mathcal{M}} = \{c_1, \ldots, c_h\}$ (see, for example, [1] §4.3).

Now consider just the card constraints in R. Since constraints about a sort impose no restrictions on the other sorts in FCC, the FCC solver can look at them separately by sort. So let S be one of the sorts in R and let $K = \{\neg \mathsf{card}_{S,i}\}_{i \in I} \cup \{\mathsf{card}_{S,j}\}_{j \in J}$ collect all the card constraints in R for S. Observe that K is satisfiable in FCC iff $I = \emptyset$ or $J = \emptyset$ or $\max(I) < \min(J)$. When K is satisfiable and J is non-empty, the FCC solver needs to check that $R \cup \mathsf{card}_{S,k}$ is satisfiable with $k = \min(J)$. This is immediate if $k \geq h$ where h is the number of equivalence classes the EUF solver has computed for S. Otherwise, $R \cup \mathsf{card}_{S,k}$ is satisfiable if and only if enough of those classes can be merged to reduce their number to at most k. At this point the FCC solver needs to strengthen R with one equality $c \approx d$ between distinct representative constants. Since $R \cup \{c \approx d\}$ may be unsatisfiable in FCC, or $G \cup \{c \approx d\}$ may be unsatisfiable in EUF, all possible equalities between representative constants may have to be considered. If none of them works, $G \cup \mathsf{card}_{S,k}$ is unsatisfiable. Otherwise, R must be strengthened again as above until S has at most k equivalence classes.

Disequality Graphs. Following the splitting-on-demand approach, the FCC solver will return "satisfiable" or "unsatisfiable" only if it can determine the satisfiability of its constraints without having to guess any equality between representative constants. Otherwise, it will simply identify a possible equality $c \approx d$ and let the SAT engine decide on it by returning the *merge lemma* $c \approx d \vee c \not\approx d$.[6] The solver is able to reduce the number of equality guesses by maintaining a *disequality graph* for each free sort S. This is an undirected graph whose vertices are representative constants for S that occur in G, and whose edges link only vertices c, d with $G \models_{\text{EUF}} c \not\approx d$. This data structure tells the FCC solver that certain pairs of constants, the linked ones, cannot be equated.

We illustrate the overall mechanism and the intended collaboration dynamics between the EUF and the FCC solver with a couple of examples.

Example 1. Let $G \cup R$ be $\{a \approx f(b), b \approx f(c), a \not\approx b, b \not\approx c, \mathsf{card}_{S,2}\}$ over the single sort S. From it, the EUF solver computes the congruence $\{\{a, f(b)\}, \{b, f(c)\}, \{c\}\}$. Using a, b, c as the representatives, the FCC solver builds the disequality graph with edges $\{(a,b), (b,c)\}$. Since $\mathsf{card}_{S,2}$ limits the size of S to at most 2, the FCC solver generates the merge lemma $a \approx c \vee a \not\approx c$. Strengthening R with $a \approx c$ produces no EUF conflicts and allows the FCC solver to answer "satisfiable".

Example 2. Consider the constraints $\{c_1 \approx c, c_4 \approx c, c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4, c_4 \not\approx c_5, \mathsf{card}_{S,2}\}$ where all the constants have sort S. The corresponding disequality graph for these constraints contains a clique of size 3. By discovering that clique, the FCC solver can conclude that it is impossible to shrink the model to 2 elements, and report a

[6] This is slightly inaccurate. In reality, the solver asks the SAT engine to apply **Decide** on $c \approx d$.

conflicting clause consisting of the literals that explain the unsatisfiability: $c_1 \not\approx c \vee c_4 \not\approx c \vee c_1 \approx c_2 \vee c_2 \approx c_3 \vee c_3 \approx c_4 \vee \neg \mathsf{card}_{S,2}$.

3.2 FCC **Solver Enhancements: Regions**

The FCC solver is able to detect the unsatisfiability of a constraint set $D \cup \{\mathsf{card}_{S,k}\}$, where D is a set of disequalities between representative constants for S, only if the disequality graph corresponding to D contains a $(k+1)$-vertex clique. Now, even just checking for the presence of a $(k+1)$-clique in a n-vertex graph is too expensive in general—as its worst case complexity is $O(n^{k+1}(k+1)^2)$. We have developed a method that reduces that cost in practice by partitioning the vertices of the graph into *regions*.

The partition is updated incrementally as the graph evolves so as to maintain the invariant that any $(k+1)$-clique of the graph is entirely contained in one of the regions. We call such partitions a *regionalization* of the graph. Regionalizations help provide a *weak effort* type of satisfiability check where the theory solver reports "(un)satisfiable" only when the (un)satisfiability of its constraints is immediate, and reports "unknown" otherwise. Frequent weak effort checks are commonly used in SMT solvers [12]. They are useful during the extension of the assignment M, to avoid extensions that are clearly unsatisfiable in one of theories. In contrast, *strong effort checks*, where the theory solver is required to give a definite answer or provide a splitting lemma, are needed (for correctness) only when the SMT solver has found an assignment M that propositionally entails the current set of ground clauses.

Weak Effort Checks. When a vertex is added to the graph it starts into its own singleton region. Two regions are combined into one whenever the addition of an edge or the merging of two nodes breaks the regionalization invariant by creating too many *inter-regional* edges, which link two vertices belonging to different regions. The choice of which regions to combine is made heuristically in an effort to increase the likelihood of generating a $(k+1)$-clique. Specifically, a region is combined with the one with which it shares the highest number of inter-regional edges. For any given regionalization of the disequality graph, *small regions*, those with less than $k+1$ vertices, cannot give rise to a clique. So our solver ignores them and focuses on the *large* regions, the other ones. The solver maintains a set of $k+1$ *watched vertices* from each large region, representing a candidate clique. A new vertex from the region is added to this set whenever two vertices in it are merged, to maintain the set's size at $k+1$.[7] The solver also keeps track of all pairs of watched vertices that are not linked. This way it knows that the region contains a $(k+1)$-clique as soon as the set of those pairs becomes empty. Similarly, it knows that the graph cannot contain any $(k+1)$-cliques as soon as the set of regions reduces to one small region. These facts are used to determine the answer of weak effort satisfiability checks.

Example 3. Consider the constraints $\{c_1 \not\approx c_2, c_2 \not\approx c_3, c_3 \not\approx c_4, \mathsf{card}_{S,2}\}$, all over sort S, and the partition $\{\{c_1, c_2\}, \{c_3, c_4\}\}$. That partition is indeed a regionalization because a 3-clique can span two regions only if it contains two interregional edges, and this

[7] If there are no new vertices to watch, the region has become small and can be ignored.

partition only has one. Adding the disequation $c_2 \not\approx c_4$ or $c_1 \not\approx c_4$, say, breaks the regionalization invariant. In either case, FCC the solver would then merge the two regions (and discover a 3-clique with nodes c_2, c_3, c_4 in the first case).

Strong Effort Checks. When the satisfiability of the current set of FCC constraints cannot be determined immediately as in weak effort checks, the FCC solver looks at ways to reduce the size of large regions by guessing an equality between two unlinked watched vertices in the same region, and returning the corresponding merge lemma. If there are no large regions, it creates one by combining smaller ones heuristically. This process eventually leads to the creation of a clique of watched vertices contradicting the corresponding cardinality constraint, or the creation of a single small region per sort. The solver can then report unsatisfiability in the first case and satisfiability in the second.

3.3 A Finite Model Finding Strategy

In this subsection we show how a DPLL$(T_1, \ldots, T_m)$ solver can be turned into a finite model finder for quantified formulas by incorporating the FCC solver described above and adopting a specific execution strategy.

Suppose T_1 is EUF and T_2 is FCC. The strategy starts by applying the **Propagate**$_i$ rules as much as possible. If propositional or theory conflicts arise in the process, an application of **Conflict**$_i$ and then **Fail** ends the execution. Otherwise, for each free sort S in the input problem, the SMT solver asks the FCC solver for a *cardinality lemma* $\mathsf{card}_{S,k_S} \vee \neg\mathsf{card}_{S,k_S}$, encoding the guessing of a concrete cardinality bound on S and corresponding to one application of **Learn**$_2$. This is followed by a corresponding number of **Decide** applications, asserting the cardinality constraint card_{S,k_s} from each of the new lemmas. From then on, the execution proceeds as usual in DPLL$(T_1, \ldots, T_m)$ solvers, but with no applications of the $\forall$**-Inst** rule. (The $\exists$**-Inst** rule is applied, once, as soon as the proxy literal $\neg a$ gets added to the current assignment.)

In particular, the EUF solver computes the congruence closure of all the EUF equalities in the assignment M and adds to it, with **Propagate**$_1$, all the entailed equalities it derives. The FCC solver builds each disequality graph incrementally, starting with one whose vertices are all the representative constants of the initial EUF equivalence classes. New edges, and possibly new vertices, are added to the graph as disequalities get added to M. Vertices are merged, with each resulting vertex inheriting all the edges of the vertices it replaces, as their equivalence classes get merged by EUF and the corresponding equation between the class representatives is added to M.

As card literals and (dis)equalities between representative constants are added to M and sent to the FCC solver, the solver is asked about the satisfiability of its updated set of constraints. It answers unsatisfiable only if its card constraints are unsatisfiable or they contain a constraint of the form $\mathsf{card}_{S,k}$ and the disequality graph for S contains a $(k+1)$-vertex clique over the representative constants for S. The solver's explanation for the unsatisfiability in the latter case is a lemma of the form $\bigvee_{c \approx d \in E} \neg(c \approx d) \vee \bigvee_{\neg(c \approx d) \in D} c \approx d \vee \neg\mathsf{card}_{S,k}$ where E and D are respectively a set of equalities and disequalities in the current assignment M that together generate the clique. Returning that clause corresponds to an application of **Conflict**$_2$, which might lead, through applications of **Explain**$_i$ and then **Backjump**, to alternative identifications of

representative constants. If no other alternatives exist, **Backjump** will backtrack to the the decision level right before the addition of $\mathsf{card}_{S,k}$ to M, and assert $\neg\mathsf{card}_{S,k}$. This will cause the FCC solver to consider a larger cardinality k' for S, and return the lemma $\mathsf{card}_{S,k'} \vee \neg\mathsf{card}_{S,k'}$ when queried again.

The constraints given to the FCC solver may be unsatisfiable in the presence of $\mathsf{card}_{S,k}$ even if the disequality graph for S contains no $(k+1)$-cliques—as long as this graph has more than k nodes. To see this, simply observe that satisfiability in this case is akin to the k-colorability problem for that graph, which means that further internal search might be needed to determine satisfiability. As discussed earlier, to keep the FCC solver simple it is enough to have it just determine which pairs c,d of representative constants could be in principle identified in order to shrink the graph, and report that possibility as the merge lemma $c \approx d \vee c \not\approx d$. With a decision strategy that prefers $c \approx d$ over $c \not\approx d$, this lets the SMT solver assert $c \approx d$ with **Decide**. The new literal is then used by the FCC solver to shrink the graph and by the EUF solver to merge the equivalence classes of c and d. Unless this leads to a conflict, the FCC will continue generating merge lemmas until the size of the disequality graph for S goes down to k.

Remark 1. Because of congruence constraints, guesses on merge lemmas may sometimes lead to inconsistencies in EUF, instead of FCC, unless the EUF solver computes and propagates all entailed disequalities—which is usually not the case, for efficiency reasons. For example, suppose the current assignment M is $\{c_3 \approx f(c_1), c_4 \approx f(c_2), c_3 \not\approx c_4, \mathsf{card}_{S,2}\}$ where all the terms have sort S. Unless the EUF solver propagates the entailed literal $c_1 \not\approx c_2$, the FCC solver will construct for S the disequality graph $(\{c_1,\ldots,c_4\},\{(c_3,c_4)\})$ and may ask the SAT engine to guess $c_1 \approx c_2$. The subset $\{c_3 \approx f(c_1), c_4 \approx f(c_2), c_3 \not\approx c_4, c_1 \approx c_2\}$ of the new assignment will then be found unsatisfiable by the EUF solver. In contrast, guessing for instance $c_1 \approx c_3$ and $c_2 \approx c_4$ will produce a model of the required cardinality. □

When M propositionally entails all the ground clauses in F and all the sub-solvers (including the FCC solver) have reported their constraints to be satisfiable, those clauses have a model that interprets each free sort S as a finite set of representative constants. Following that, the SMT solver goes into an *instantiation round* where it applies the $\forall$**-Inst** rule exhaustively. That is, if for each $a \in \mathsf{M}$ and $a \Leftrightarrow \forall \mathbf{x}\, C[\mathbf{x}] \in \mathsf{F}$ it adds to F all possible well-sorted instances $C[\mathbf{c}]$, where where the elements of $\mathbf{c}$ are current representative constants. The system processes the new ground clauses, and the new constraints they generate, as described above until **Fail** is applicable or a new model of the (extended set of) ground clauses of F is found. Then the system starts another instantiation round, but adding to F only clause instances $C[\mathbf{c}]$ that had not been added in previous instantiation rounds. More precisely, it will discard a newly generated instance $C[\mathbf{d}]$ if F contains an instance $C[\mathbf{c}]$ where $\mathbf{c}$ and $\mathbf{d}$ are equivalent in the current congruence closure. The whole process stops if a new instantiation round produces no new instances. This is because at that point the literals of M have a finite model that satisfies all the instances of the quantified formulas.

In our default strategy, the cardinality upper bounds expressed with the card constraints start at 1 for each sort and are incremented only by 1 at a time, with the goal of minimizing sort sizes in candidate models. Keeping those sizes small is essential to contain the explosion of formula instances added during instantiation rounds. An additional

way to control this explosion is to replace exhaustive instantiation with smarter heuristics that avoid the generation of *redundant* instances of quantified formulas—intuitively, instances guaranteed to be satisfied by the current candidate model. We have developed an effective notion of redundancy and a non-exhaustive instantiation method that relies on advanced data structures to represent and query candidate models. That work and the performance improvements it yields are discussed in a companion paper [14].

Correctness. To argue about the soundness of our finite model finding method (i.e., the input problem is unsatisfiable whenever the execution ends with the fail state), observe first that the method can be described entirely as a particular execution strategy of the abstract framework presented in Section 2. Thus, it suffices to show that the FCC solver is itself sound.

Proposition 1. *Whenever the* FCC *solver returns "unsatisfiable" for a set R of* FCC *constraints, R is unsatisfiable in the* FCC *theory.*[8]

Our model finding method is non-terminating in general because there exists satisfiable quantified EUF formulas with no finite models. The more interesting question is whether it is *finite-model complete*, that is, guaranteed to find a finite model of the input problem if one exists. As described here, our approach is finite-model complete for input problems whose quantifiers all range over the same free sort, but not in the more general case involving several free sorts. Informally, the reason for this incompleteness is that our method allows executions that keep increasing indefinitely the size of the wrong sort. We are working on the definition of a *practical* fairness restriction on executions that addresses this issue. For the sort of applications we have been targeting, however, this source of finite-model incompleteness did not seem to be a problem.

4 Experimental Results

We implemented a finite model finder as described here within CVC4 [2] which is based on DPLL($T_1, \ldots, T_m$). We added to CVC4 our FCC solver, implementing it as a direct extension of CVC4's EUF solver. That solver maintains backtrackable data structures for representing the current congruence closure over EUF terms as well as keeping track of asserted disequalities between them. In addition, the FCC solver maintains data structures for the current regionalization as described in Section 3.2.

We ran two sets of experiments, the first to evaluate the relative effectiveness of various strategies for the FCC solver, and the second to evaluate the model finder's overall performance when used with quantified formulas. For the second set of experiments, we compared our model finder against various state-of-the-art SMT solvers, including CVC4 itself. All experiments were run on a Linux machine with an 8-core 2.60GHz Intel® Xeon® E5-2670 processor.[9]

[8] We omit the proof of this proposition because it is relatively straightforward thanks to the restricted cases in which the FCC solver returns "unsatisfiable."

[9] The finite model finder, detailed results, and the non-proprietary benchmarks discussed in this section are available at `http://cvc4.cs.nyu.edu/experiments/CAV-2013/`

4.1 FCC **Solver Evaluation**

We tested various configurations of the FCC solver, starting with the default configuration **cvc4+f**, which implements the region-based enhancements described in Section 3.2 as well as an additional enhancement where conflict clauses have simply the form $\neg\mathsf{distinct}(c_1,\ldots,c_{k+1}) \vee \neg\mathsf{card}_{S,k}$, where distinct is a variadic logical predicate satisfied exactly when its arguments evaluate to pairwise distinct elements. We also tested a configuration, **cvc4+fe**, where conflict clauses are as described in Section 3.3. This configuration avoids the introduction of new predicates into the search (the distinct ones), but has the disadvantage that it can generate different conflict clauses for essentially the same clique. Additionally, we considered configuration **cvc4+f-r**, which differs from **cvc4+f** only in that regionalizations have always just one region per sort S, encompassing the entire disequality graph for S.

We also evaluated the MACE-style approach to finite model finding described in related work, which we encoded in the configuration **cvc4+mace**. For a basic idea of this encoding in the simple case of a ground EUF formula φ involving a single sort, if $\mathbf{T}_\varphi$ is the set of all terms in φ and $c_1,\ldots,c_k$ are fresh constants serving as domain constants, this configuration uses CVC4 to check the satisfiability of

$$\varphi \wedge \mathsf{distinct}(c_1,\ldots,c_k) \wedge \bigwedge_{t\in\mathbf{T}_\varphi} (t = c_1 \vee \ldots \vee t = c_k) \tag{1}$$

for $k = 1,2,\ldots$ until (1) is found satisfiable for some k. Then, the minimal model size for φ is k. As we mentioned in Section 1, a major shortcoming of this approach is the introduction of unwanted value symmetries in the problem. CVC4 can address this issue to some extent since it incorporates a few symmetry breaking techniques directly at the EUF level [7].

We considered satisfiable benchmarks encoding randomly generated graph coloring problems and consisting of a conjunction of disequalities between constants of a single sort. In particular, we considered a total of 793 non-trivial problems containing between 20 and 50 unique constants and between 100 and 900 disequalities, and measured the time it takes each configuration to find a model of minimum size, with a 60 second timeout. For the benchmarks we tested, the configuration **cvc4+f** solves the most benchmarks, 723, within the time limit. Although not shown here in detail, **cvc4+f** was an order of magnitude faster than **cvc4+fe** on most benchmarks, with the latter only being able to solve 309 benchmarks within the time limit. This strongly suggests that generating explanations for cliques in conflict lemmas involving cardinality constraints is not an effective approach in this scheme.

Figure 2 compares the performance of the configuration **cvc4+f** against **cvc4+f-r** and **cvc4+mace**. The first scatter plot clearly shows that the **cvc4+f** configuration generally requires less time and solves more benchmarks (723 vs. 664) than **cvc4+f-r**, confirming the usefulness of a region-based approach for clique detection. The second scatter plot compares **cvc4+f** against **cvc4+mace**. The latter configuration was able to solve only 617 benchmarks and generally performed poorly on benchmarks with larger model size. The median model size of the 123 benchmarks solved only by **cvc4+f** was 17, whereas the median size of the 13 benchmarks solved only by **cvc4+mace** was 10. This suggests

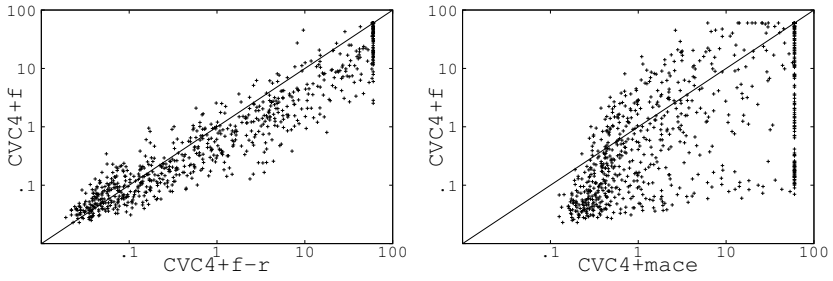


Fig. 2. Results for randomly generated benchmarks. Runtimes are in seconds, on a log-log scale.

that for larger cardinalities **cvc4+mace** suffers from the model symmetries created by the introduction of domain constants, something that **cvc4+f** avoids.

4.2 Finite Model Finder Evaluation

We also evaluated the overall effectiveness of CVC4's finite model finder for quantified SMT formulas. We used benchmarks derived from verification conditions generated by DVF [10], a tool used at Intel for verifying properties of security protocols and design architectures, among other applications. Both unsatisfiable and satisfiable benchmarks were produced, the latter by manually removing necessary assumptions from verification conditions. All benchmarks contain quantifiers, although only over free sorts, and span a wide range of theories, including linear integer arithmetic, arrays, EUF, and inductive datatypes.

For comparison we looked at the SMT solvers CVC3 (version 2.4.1), Yices (version 1.0.32), Z3 (version 4.1). We also considered CVC4 (release r4751) in native mode, that is, without the finite model finding techniques described here. We did not look at traditional theorem provers and finite model finders because they do not have built-in support for the theories in our benchmark set. All the solvers considered use E-matching as a heuristic method for answering unsatisfiable in the presence of universally quantified formulas. CVC4 uses no sophisticated techniques for detecting satisfiability, which means that it reports "unknown" for most satisfiable quantified problems. In contrast, Z3 additionally relies on model-based quantifier instantiation [9] to be able to detect satisfiable quantified problems in certain cases.

The results, separated into unsatisfiable and satisfiable instances, are shown in Figure 3 for five classes of benchmarks and a timeout of 600s per benchmark. The first two classes, **refcount** and **german**, represent verification conditions for systems described in [10]; benchmarks in the third are taken from [17]; the last two classes are verification problems internal to Intel.

For the satisfiable benchmarks, our finite model finder is the only tool capable of solving any instance in the last three benchmark classes.[10] In fact, **cvc4+f** is able to solve all but two, and most of them in less than a second. By comparing **cvc4+f** against

[10] Yices reports "unsat" for two of these benchmarks. We believe that, based on the way they were constructed, the two benchmarks are in fact satisfiable. Also, all other solvers (including previous versions of Yices) time out or answer "unknown".

Sat	**german** (45)		**refcount** (6)		**agree** (42)		**apg** (19)		**bmk** (37)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	0	0.0	0	0.0	0	0.0	0	0.0	0	0.0
yices	2	0.02	0	0.0	0	0.0	0	0.0	0	0.0
z3	45	1.1	1	7.0	0	0.0	0	0.0	0	0.0
cvc4	2	0.00	0	0.00	0	0.0	0	0.0	0	0.0
cvc4+f	**45**	0.3	**6**	0.1	**42**	15.5	**18**	200.0	**36**	1201.5
cvc4+f-r	**45**	0.3	**6**	0.1	42	18.6	15	364.3	34	720.4

Unsat	**german** (145)		**refcount** (40)		**agree** (488)		**apg** (304)		**bmk** (244)	
	solved	time	solved	time	solved	time	solved	time	solved	time
cvc3	145	0.4	**40**	0.2	457	6.8	267	77.0	229	76.2
yices	145	1.8	40	7.0	488	1475.4	304	35.8	244	25.3
z3	145	1.9	40	0.9	**488**	10.6	304	12.2	244	5.3
cvc4	**145**	0.1	**40**	0.2	484	6.8	**304**	11.2	**244**	2.9
cvc4+f	145	0.8	40	0.4	476	3782.1	298	2252.5	242	1507.0
cvc4+f-r	145	0.4	**40**	0.2	475	1574.3	294	3836.0	240	1930.5

Fig. 3. Results for DVF benchmarks. All runtimes are in seconds.

cvc4+f-r, we see that the region-based approach for recognizing cliques is beneficial, particularly for the harder classes where the latter configuration solves fewer benchmarks within the timeout. The model sizes found for these benchmarks were relatively small, only a handful had a model with sort cardinalities larger than 4. To our knowledge, our model finder is the only tool capable of solving these benchmarks.

For the unsatisfiable benchmarks, Yices and Z3 can solve all of them, with Z3 being much faster in some cases. Interestingly, all of these benchmarks are solved in less than 3s by either **cvc4** (plain CVC4) or **cvc4+f**, indicating that a combination of the two is advantageous in general. We observe that **cvc4+f** is orders of magnitude slower than the SMT solvers on these benchmarks. This is, however, to be expected since it is geared towards finding models, and applies exhaustive instantiation with increasingly large cardinality bounds, which normally delays the discovery that the problem is unsatisfiable regardless of those bounds.

5 Conclusion and Further Work

We presented a method for endowing DPLL(T)-based SMT solvers with finite model finding capabilities for quantified SMT formulas with quantifiers ranging over free sorts. The method relies on a novel and efficient sub-solver for finite cardinality constraints that is fully integrated in the overall SMT solver. Our experimental results with benchmarks generated from a variety of verification applications show that our model finding approach is superior to current quantifier instantiation methods in SMT in the case of satisfiable inputs.

Future work will focus on identifying suitable fair execution strategies that guarantee finite model completeness for problems with multiple free sorts. We are also plan to investigate further approaches for finding models of formulas with quantifiers ranging also over built-in domains such as the integers.

Acknowledgments. We thank Clark Barrett, Morgan Deters, Dejan Jovanović, and François Bobot for their suggestions and assistance in the implementation of our model

finder. We thank Mark Tuttle for providing the model from which the **agree** benchmarks were generated.

References

[1] Baader, F., Nipkow, T.: Term Rewriting and All That. Cambridge University Press (1998)
[2] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
[3] Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in SAT modulo theories. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 512–526. Springer, Heidelberg (2006)
[4] Baumgartner, P., Fuchs, A., de Nivelle, H., Tinelli, C.: Computing finite models by reduction to function-free clause logic. Journal of Applied Logic 7, 58–74 (2009)
[5] Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: Delayed theory combination vs. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. Nelson-Oppen for satisfiability modulo theories: a comparative analysis. AMAI 55(1-2), 63–99 (2009)
[6] Claessen, K., Sörensson, N.: New techniques that improve MACE-style finite model building. In: CADE-19 Workshop: Model Computation – Principles, Algorithms, Applications, pp. 11–27 (2003)
[7] Déharbe, D., Fontaine, P., Merz, S., Woltzenlogel Paleo, B.: Exploiting symmetry in SMT problems. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 222–236. Springer, Heidelberg (2011)
[8] Ge, Y., Barrett, C., Tinelli, C.: Solving quantified verification conditions using satisfiability modulo theories. AMAI 55(1-2), 101–122 (2009)
[9] Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
[10] Goel, A., Krstić, S., Tuttle, R.L.M.: SMT-based system verification with DVF. In: Proceedings of SMT 2012 (2012)
[11] Krstić, S., Goel, A.: Architecting solvers for SAT modulo theories: Nelson-Oppen with DPLL. In: Konev, B., Wolter, F. (eds.) FroCos 2007. LNCS (LNAI), vol. 4720, pp. 1–27. Springer, Heidelberg (2007)
[12] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: from an abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). Journal of the ACM 53(6), 937–977 (2006)
[13] Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005)
[14] Reynolds, A., Tinelli, C., Goel, A., Krstić, S., Deters, M., Barrett, C.: Quantifier instantiation techniques for finite model finding in SMT. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 377–391. Springer, Heidelberg (2013)
[15] Tinelli, C., Harandi, M.T.: A new correctness proof of the Nelson–Oppen combination procedure. In: Proceeding of FroCoS 1996, pp. 103–120. Kluwer (1996)
[16] Torlak, E., Jackson, D.: Kodkod: a relational model finder. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 632–647. Springer, Heidelberg (2007)
[17] Tuttle, M.R., Goel, A.: Protocol proof checking simplified with SMT. In: Proceedings of NCA 2012, pp. 195–202. IEEE Computer Society (2012)
[18] Zhang, J., Zhang, H.: SEM: a system for enumerating models. In: Proceedings of IJCAI 1995, pp. 298–303 (1995)

JBernstein: A Validity Checker for Generalized Polynomial Constraints

Chih-Hong Cheng[1], Harald Ruess[1], and Natarajan Shankar[2]

[1] fortiss GmbH, Guerickestr. 25, 80805 München, Germany
{cheng,ruess}@fortiss.org
[2] SRI International, 333 Ravenswood, Menlo Park, CA 94025, United States
shankar@csl.sri.com

1 Overview

Efficient and scalable verification of nonlinear real arithmetic constraints is essential in many automated verification and synthesis tasks for hybrid systems, control algorithms, digital signal processors, and mixed analog/digital circuits. Despite substantial advances in verification technology, complexity issues with classical decision procedures for nonlinear real arithmetic are still a major obstacle for formal verification of real-world applications.

Recently, Muñoz and Narkawicz [3] proposed a procedure for deciding the validity of quantifier-free nonlinear real arithmetic based on the well-known transformation to Bernstein polynomials. Their Kodiak system outperforms tools based on cylindric algebraic decomposition, including QEPCAD [1] and REDLOG [2], in many cases.

Starting from the algorithms by Muñoz and Narkawicz [3] we extended the approach and implemented the (little) verification engine JBernstein, which checks the validity of finite conjunctions of nonlinear constraints of the form

$$\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \\ (\textstyle\bigwedge_{j=1}^{n} P_j(x_0, \ldots, x_m) \prec_j c_j) \;\rightarrow\; Q(x_0, \ldots, x_m) \prec d$$

P_j and Q are real polynomials over the variables x_0 through x_m, each x_i is interpreted over closed intervals $[l_i, u_i]$ with real-valued lower and upper bounds, c_j and d are real-valued constants, and the symbols $\prec_j$ and $\prec$ are arithmetic inequalities in $\{>, \geq, <, \leq\}$. These constraints support *assume-guarantee* style of reasoning about open systems, with P_i the assumptions on the environment and Q the corresponding guarantee of the system under consideration.

The Java implementation JBernstein includes a number of algorithmic optimizations as described in Section 2, for example, for avoiding unnecessary case splits. In particular, JBernstein uses double-precision floating-point arithmetic of Java in a sound way via constraint strengthening (Section 3).

The resulting runtimes of JBernstein are compared with those reported by Muñoz and Narkawicz [3] for their Kodiak implementation, QEPCAD, and REDLOG. This comparison uses the PVS test suite as compiled by Muñoz and Narkawicz [3]. The experimental evaluation of these optimizations indicates that for complex problems, JBernstein is usually an order of magnitude faster than earlier results by Muñoz and

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 656–661, 2013.

Table 1. Performance of JBernstein and other tools. Results from other tools are taken from [3]. The measured unit for execution time is in milliseconds.

Problem	JBernstein	Kodiak [3]	$REDLOG_{rlqe}$ [3]	$REDLOG_{rlcad}$ [3]	QEPCAD [3]	Metit [3]
Schwefel (∀)	159	940	490	> 300000	840	110
Schwefel (∃)	126	280	138900	> 300000	910	(n/a)
Reaction Diffusion (∀)	7	< 10	340	370	10	90
Reaction Diffusion (∃)	3	< 10	340	350	10	(n/a)
Caprasse (∀)	23	290	1750	> 300000	6540	160
Caprasse (∃)	8	310	15060	> 300000	6540	(n/a)
Lotka-Volterra (∀)	5	100	360	450	10	100
Lotka-Volterra (∃)	4	< 10	350	400	10	(n/a)
Butcher (∀)	19	200	420	> 300000	(abort)	(abort)
Butcher (∃)	65	200	360	> 300000	(abort)	(n/a)
Magnetism (∀)	125	73540	670	360	180	540
Magnetism (∃)	115	320	420	360	350	(n/a)
Heart Dipole (∀)	460	7360	> 300000	> 300000	> 300000	> 300000
Heart Dipole (∃)	405	3700	> 300000	> 300000	> 300000	(n/a)

§ For this example, we only ask > -0.0001, as precision can not be maintained for the original property.
‡ For this example, we only ask > 0.0001.

Narkawicz or our un-optimized implementation, and it is two orders of magnitude faster than QEPCAD or REDLOG. For the experimental evaluation we use a similar hardware setting (Intel Core Duo 2.4 Ghz, MacOS, 8 GB RAM) as reported in [3]. The optimization power comes with harder problems, as with these problems refinements are used heavily, and the accumulative effect of optimization comes. These initial experimental results are indeed promising, but, clearly, an extended and improved set of benchmarks is needed to obtain an indication about the asymptotic behavior of these solvers.

JBernstein is implemented in Java without further dependencies. It is freely available under the LGPL version 3 license at

`http://sourceforge.net/projects/jbernstein/`

2 Algorithmic Optimizations

We outline the algorithm of Muñoz and Narkawicz in [3] and the optimizations thereof in JBernstein using simplified constraints of the form

$$\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \phi(x_0, \ldots, x_m) > c\,.$$

The Bernstein approach first performs a *range-preserving* transformation from $[a, b]$ to $[0, 1]$ to obtain a constraint of the form

$$\forall y_0 \in [0, 1], \ldots, y_m \in [0, 1] : \phi'(y_0, \ldots y_m) > c\,,$$

such that $x_i = l_i + y_i(u_i - l_i)$ for all $i \in \{1, \ldots, m\}$ and $\phi(x_1, \ldots, x_m) = \phi'(y_1, \ldots, y_m)$. The polynomial ϕ' is then translated from *polynomial basis* into *Bernstein basis*. For example, $\phi'(y) = 4y^2 - 4y + 1$ has polynomial basis $\{y^2, y, 1\}$. It is rewritten as $\mathbf{1}\binom{0}{2}(1-y)^2$-$\mathbf{2}\binom{1}{2}y(1-y) + \mathbf{1}\binom{2}{2}(y)^2$, with $\{\binom{k}{2}y^k(1-y)^{2-k} | k = 0, 1, 2\}$ being the Bernstein basis. If all coefficients of the polynomial represented in Bernstein basis are greater than c, then the original constraint holds (true). Otherwise one checks if there exists a coefficient of an *endindex* (Bernstein basis vector where every term is either 0

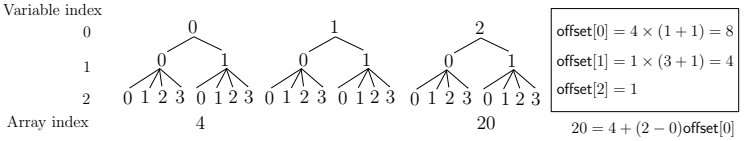


Fig. 1. Index shifting by table look-up

or of highest degree) that is smaller or equal to c. For example, for $\forall y \in [0,1] : \phi'(y) > -3$, as all Bernstein coefficients $1, -2, 1$ are greater than -3, and the property holds. If the property is $\forall y \in [0,1] : \phi'(y) > 2$, as for the first and the last coefficient (in $\{\binom{n}{2} y^k (1-y)^{2-n} | n = 0, 1, 2\}$, $n = 0, 2$ are endindices), we have $1 \not> 2$, and the property fails to hold at $y = 0$ and $y = 1$ (false).

Lazy refinement. The checking process returns unknown if neither of these two conditions holds, i.e., there exists some non-endindex coefficients less or equal to c. In these cases, the algorithm does range splitting on some chosen variable from $[0,1]$ to $[0, \frac{1}{2}]$ and $[\frac{1}{2}, 1]$, it generates Bernstein polynomials for each subspace, and it checks if the property does indeed hold for both polynomials. The Bernstein polynomial checker refines a subspace only when a decisive proof has not been found for values within this subspace.

Inputs of the form $\rho > d \rightarrow \varphi > e$ are not considered within the paper by Muñoz and Narkawicz. In JBernstein, such an input is rewritten to its disjunctive form $\rho \leq d \vee \varphi > e$. The checker returns true if one of them is valid, and it returns false if there exists an endindex of both polynomials whose coefficients violate the constraint separately.

Recursive variable exploration. The refinement process involves recursive calls of variable selection and domain partitioning to generate new Bernstein polynomials. For efficiency it is important to avoid variable selections which lead to the generating of an excessive number of polynomials for which the checker returns unknown. As the algorithm is recursive, a naïve deterministic implementation can, given a fixed recursion depth, try to refine the first variable continuously until the end of the recursion depth. It turns out that this strategy often leads to a generation of numerous useless Bernstein polynomials without providing definite proofs. The reason is explained using the following extreme case. Continuously refine only one variable, say y_0, then eventually the generated Bernstein polynomial will have y_0 with its domain converged from $[0,1]$ to one single point $\alpha \in [0,1]$. This means that in the original polynomial, the refinement tries to analyze the polynomial by setting x_0 to $l_0 + \alpha(u_0 - l_0)$, whose proving process is, intuitively, nearly as hard as working on the original polynomial. Given a certain budget for maximum allowed recursion, first trying deep recursion over one particular variable is therefore, unlikely to succeed in most cases. Therefore, the implementation of JBernstein uses a round-robin selection strategy on the variables during recursion to avoid worst case scenarios.

Avoiding superfluous computations. The third aspect concerning efficiency focuses on reducing the unit cost for generating every new Bernstein polynomial and checking the property. Generating a new Bernstein polynomial (domain partition) entails generating

all of its coefficients, which themselves are derived from the coefficients of the original Bernstein polynomial (without domain partition). The new coefficients $b^L_{\mathbf{k}}$ for the left partitioned polynomial (for the right polynomial, the formula is similar), where y_j is chosen for partition, is achieved using the following formula [3].

$$b^L_{\mathbf{k}} = \sum_{r=0}^{k_j} \frac{1}{2^{k_j}} \binom{k_j}{r} b_{\mathbf{k}\ with\ [j:=r]} \tag{1}$$

In Eq. 1, for an m-variate polynomial, $\mathbf{k}$ is a vector of m tuples of positive integers or 0, where each term is smaller than the cardinality (i.e., every $\mathbf{k}$ is the unique signature of each Bernstein basis vector). k_j is the j-th value of $\mathbf{k}$, and $b_{\mathbf{k}\ with\ [j:=r]}$ is the coefficient of the original Bernstein polynomial, where "$\mathbf{k}\ with\ [j := r]$" is an m-tuple that is equal to $\mathbf{k}$ on every index, except in index j where it has value r. The following optimizations are used by JBernstein:

1. Factor out the constant $\frac{1}{2^{k_j}}$ from summation.
2. Replace the computation of $\binom{k_j}{r}$ by table look-up.
3. Compute $b_{\mathbf{k}\ with\ [j:=r]}$ in an optimized way as follows:
   - Statically store $\mathbf{k}$ linearly in an array (prior to the range transformation). Figure 1 indicates how vectors are arranged. E.g., when $\mathbf{k} = (0, 1, 0)$, it is located in index 4. Store each $b_{\mathbf{k}}$ linearly in an array following the above order, for every Bernstein polynomial.
   - When the solver iterates the array to create $b^L_{\mathbf{k}}$, the index of $\mathbf{k}$ is known.
   - To find $b_{\mathbf{k}\ with\ [j:=r]}$, it amounts to finding the index of "$\mathbf{k}\ with\ [j := r]$". Due to our formulation, it can now be translated to an index offset problem: given the index of $\mathbf{k}$, what is the offset when replacing the j-th index with value r? The offset is $(r - k_j)$offset$[j]$, where offset$[j] = (Deg_{j+1} + 1) \times \ldots \times (Deg_{m-1}+1)$, where Deg_s is the highest degree that appears for variable v_s in the polynomial. The array offset can also be computed statically prior to the range-preserving transformation. Figure 1 illustrates the computation of the index of of $(2, 1, 0)$ from $\mathbf{k} = (0, 1, 0)$.
   - Therefore, for $b_{\mathbf{k}\ with\ [j:=r]}$ the solver uses three table look-ups (for offset$[j]$, k_j and the final value), one substraction, multiplication and addition.
4. Integrate the coefficient generation and the checking process. In each Bernstein polynomial, create an internal Boolean variable field isUnknown that is initially set to false. During the construction, check if a particular coefficient satisfies the property only when it belongs to the endindex or isUnknown equals false. Once if the polynomial is diagnosed as unknown, then there is no need to check for coefficients from non-endindices. Deciding whether a certain index is an endindex is also done statically once and is replaced by table look-up in later computations.

3 Sound Usage of double

We now justify the use of double (double-precision 64-bit IEEE 754 floating point) for data representation. In JBernstein, potential errors due to imprecision of double are handled by the following methodology (for the ease of explanation, we again set the property to be $\forall x_0 \in [l_0, u_0], \ldots, x_m \in [l_m, u_m] : \phi(x_0, \ldots, x_m) > c$):

- Select a positive error-estimate ϵ such as 10^{-5}.
- To return true, in property checking all coefficients shall be greater than $c + \epsilon$.
- To return false, in property checking the solver needs to find a coefficient b_k from an endindex such that $b_k \leq c - \epsilon$.
- If neither of the above two cases holds, the solver either proceeds with domain refinement (when recursion is still allowed) or returns unknown.

The correctness relies on a crucial requirement that the accumulated error for each computed coefficient should never exceed ϵ. Instead of keeping track of the error during the computation, we apply *static analysis* on the algorithm to generate a safe error-estimate that holds for each computed coefficient, based on the polynomial constraint itself and the number of maximum refinement attempts. Due to space limits, we only review the key feature in refining a subspace to generate new Bernstein polynomials (a full text concerning the sound usage of double can be found in the technical report).

For each refinement, recall in Eq. 1 where we have $b^L_{\mathbf{k}} = \sum_{r=0}^{k_j} \frac{1}{2^{k_j}} \binom{k_j}{r} b_{\mathbf{k}\ with\ [j:=r]}$. E.g., when $k_j = 4$, $\frac{1}{2^{k_j}} \binom{k_j}{r}$ equals $\frac{1}{16}, \frac{4}{16}, \frac{6}{16}, \frac{4}{16}, \frac{1}{16}$ (the sum of these values is 1) for $r = 0, 1, 2, 3, 4$. This means that each new coefficient is a weighted sum from old coefficients. If each original coefficient $b_{\mathbf{k}\ with\ [j:=r]}$ has an error-estimate bounded by κ, in an idealized computation, the generated $b^L_{\mathbf{k}}$ also has an error-estimate bounded by κ. This gives an intuition that the growth of error should be very slow (nearly linear to the number of refinement attempts) within the refinement process. The behavior of linear growth is the key ingredient that makes our methodology applicable[1], which is one of the nice properties for Bernstein polynomials.

4 Example

For example, assume-guarantee-style constraints such as

$$\forall x_0 \in [0, 1] : (6x_0 - 1 > 0 \wedge 3x_0 - 1 < 0) \rightarrow 125x_0^3 - 175x_0^2 + 70x_0 - 8 > 0$$

are handled by the textual interface of JBernstein as described in Figure 4:

- The statement `VAR 1` in Figure 4 indicates that the constraint has one variable of name x_0, and statements such as `"BOUND x0 [0,1]"` are used to specify that x_0 is interpreted over the closed interval $[0, 1]$.
- `CONJUNCTION 1` specifies the use of one assume-guarantee rule.
- `ASSUMP 0 2` indicates that the first assume-guarantee rule (indexed 0) has two assumptions $6x_0 - 1 > 0$ and $3x_0 - 1 < 0$.
- For the first assumption (indexed 0) of the first assume-guarantee rule ($6x_0 - 1 > 0$), use "`COEF A0_0 (1) 6`", "`COEF A0_0 (0) -1`", "`SIGN A0_0 GT`", and "`VALUE A0_0 0`" to specify the polynomial.

[1] If every refinement computation brings κ twice as large, in static analysis, applying recursive expansion for small steps like 100 will make an initially small error-estimate prohibitively huge in the lastly generated Bernstein polynomial. It is also important to observe from this example that the division of 16 actually only involves the decrease of exponent by 4 in double without precision loss.

```
## FORALL x\in[0,1]: (6x-1>0 && 3x-1<0) -> (5x-1)(5x-2)(5x-4)>0

## Specify to use one variables x0
VAR 1

## Specify the NUMBER of conjunctions
## Usage: "CONJUNCTION NUMBER"
CONJUNCTION 1

## Specify the NUMBER of assumptions in the INDEX-th conjunction element
## Usage: "ASSUMP INDEX NUMBER"
ASSUMP 0 2
## Specify the coefficient of the polynomial (assumption) 5x+0>0 as
## E.g., A1_0 means the 1st assumption (indexed 0) in the 2nd conjunction (indexed 1)
## First assumption
COEF A0_0 (1) 6
COEF A0_0 (0) -1
SIGN A0_0 GT
VALUE A0_0 0

## Second assumption
COEF A0_1 (1) 3
COEF A0_1 (0) -1
SIGN A0_1 LT
VALUE A0_1 0

## Specify the coefficient of the polynomial (guarantee)
##  (5x-1)(5x-2)(5x-4) = 125x^3-175x^2+70x-8
COEF G0 (3) 125
COEF G0 (2) -175
COEF G0 (1) 70
COEF G0 (0) -8
SIGN G0 GT
VALUE G0 0

## Specify the bound for each variable
BOUND x0 [0, 1]
## Result: FALSE
```

Fig. 2. Assume-guarantee-style constraints in JBernstein

Acknowledgements. We thank Dr. César Muñoz (NASA Langley) for his support and helpful suggestions.

References

1. Brown, C.W.: QEPCAD-B: a program for computing with semi-algebraic sets using CADs. SIGSAM Bull. 37(4), 97–108 (2003)
2. Dolzmann, A., Sturm, T.: REDLOG: computer algebra meets computer logic. SIGSAM Bull. 31(2), 2–9 (1997)
3. Muñoz, C., Narkawicz, A.: Formalization of a representation of Bernstein polynomials and applications to global optimization. Journal of Automated Reasoning (2012)

ILP Modulo Theories

Panagiotis Manolios and Vasilis Papavasileiou

Northeastern University
{pete,vpap}@ccs.neu.edu

Abstract. We present Integer Linear Programming (ILP) Modulo Theories (IMT). An IMT instance is an Integer Linear Programming instance, where some symbols have interpretations in background theories. In previous work, the IMT approach has been applied to industrial synthesis and design problems with real-time constraints arising in the development of the Boeing 787. Many other problems ranging from operations research to software verification routinely involve linear constraints and optimization. Thus, a general ILP Modulo Theories framework has the potential to be widely applicable. The logical next step in the development of IMT and the main goal of this paper is to provide theoretical underpinnings. This is accomplished by means of BC(T), the Branch and Cut Modulo T abstract transition system. We show that BC(T) provides a sound and complete optimization procedure for the ILP Modulo T problem, as long as T is a decidable, stably-infinite theory. We compare a prototype of BC(T) against leading SMT solvers.

1 Introduction

The primary goal of this paper is to present the theoretical underpinnings of the Integer Linear Programming (ILP) Modulo Theories (IMT) framework for combining ILP with background theories. The motivation for developing the IMT framework comes from our previous work, where we used an ILP-based synthesis tool, CoBaSA (Component-Based System Assembly), to algorithmically synthesize architectural models using the actual production design data and constraints arising during the development of the Boeing 787 Dreamliner [16]. According to Boeing engineers, previous methods for creating architectural models required the *"cooperation of multiple teams of engineers working over long periods of time."* We were able to synthesize architectures in minutes, directly from the high-level requirements. What made this possible was the combination of ILP with a custom decision procedure for hard real-time constraints [16], *i.e.*, an instance of IMT.

ILP has been the subject of intensive research for more than five decades [13]. ILP solvers [1,2] are routinely used to solve practical optimization problems from a diverse set of fields including operations research, industrial engineering, artificial intelligence, economics, and software verification. Based on our successful use of the IMT approach to solve architectural synthesis problems and the widespread applicability of ILP and optimization, we hypothesize that IMT

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 662–677, 2013.

has the potential to enable interesting new applications, analogous to what is currently happening with Satisfiability Modulo Theories [3,10,27,8].

We introduce the theoretical underpinnings of IMT via the BC(T) framework (Branch and Cut Modulo T). BC(T) can be thought of as the IMT counterpart to the DPLL(T) architecture for lazy SMT [27]. BC(T) models the branch-and-cut family of algorithms for integer programming as an abstract transition system and allows plugging in theory solvers. Building on classical results on combining decision procedures [23,31,19], we show that BC(T) provides a sound and complete optimization procedure for the combination of ILP with stably-infinite theories. As a side-product of our theoretical study of IMT, we show how to bound variables while preserving optimality modulo the combination of Linear Integer Arithmetic and a stably-infinite theory.

The rest of the paper is organized as follows. In Section 2, we formally define IMT and provide an abstract BC(T) architecture for solving IMT problems. IMT can be seen as SMT with a more expressive core than propositional logic. We elaborate on the relationship between IMT and SMT in Section 3. We have implemented BC(T), using the SCIP MIP solver [2] as the core solver. We carried out a sequence of experiments, as outlined in Section 4. The first experiment shows that for our synthesis problems, ILP solvers [1,2] outperform the Z3 SMT solver [8]. In the second experiment, we compared our prototype implementation with state-of-the-art SMT solvers [8,14] on SMT-LIB benchmarks. An analysis of the results suggests that BC(T) is an interesting future alternative to the DPLL(T) architecture. We provide an overview of related work in Section 5 and conclude with Section 6.

2 BC(T)

In this section, we formally define IMT. We also provide a general BC(T) architecture for solving IMT problems. We describe BC(T) by means of a *transition system*, similar in spirit to DPLL(T) [27]. The BC(T) architecture allows one to obtain a solver for ILP Modulo T by combining a branch-and-cut ILP solver with a background solver for T.

2.1 Formal Preliminaries

An *integer linear expression* is a sum of the form $c_1v_1 + \cdots + c_nv_n$ for integer constants c_i and variable symbols v_i. An *integer linear constraint* is a constraint of the form $e \bowtie r$, where e is an integer linear expression, r is an integer constant, and $\bowtie$ is one of the relations $<$, $\leq$, $=$, $>$, and $\geq$. An *integer linear formula* is a set of (implicitly conjoined) integer linear constraints. We will use propositional connectives over integer linear constraints and formulas as appropriate and omit $\wedge$ when this does not cause ambiguity (*i.e.*, juxtaposition will denote conjunction). An *integer linear programming (ILP) instance* is a pair C, O, where C is an integer linear formula, and the *objective function* O is an integer linear expression. Our goal will always be *minimizing* the objective function.

We assume a fixed set of variables $\mathcal{V}$. An *integer assignment* A is a function $\mathcal{V} \to \mathbb{Z}$, where $\mathbb{Z}$ is the set of integers. We say that an assignment A *satisfies* the constraint $c = (c_1 v_1 + \cdots + c_n v_n \ \bowtie \ r)$ (where $\bowtie$ is one of the relations $<$, $\leq$, $=$, $>$, $\geq$, and every v_i is in $\mathcal{V}$) if $\sum_i c_i \cdot A(v_i) \ \bowtie \ r$. An assignment A satisfies a formula C if it satisfies every constraint $c \in C$. A formula C is *integer-satisfiable* or *integer-consistent* if there is an assignment A that satisfies C. Otherwise, it is called *integer-unsatisfiable* or *integer-inconsistent*.

A signature Σ consists of a set Σ^C of constant symbols, a set Σ^F of function symbols, a set Σ^P of predicate symbols, and a function $ar : \Sigma^F \cup \Sigma^P \to \mathbb{N}^+$ that assigns a non-zero natural number (the arity) to every function and predicate symbol. A Σ-formula is a first-order logic formula constructed using the symbols in Σ. A Σ-theory T is a closed set of Σ-formulas (*i.e.*, T contains no free variables). We will write theory in place of Σ-theory when Σ is clear from the context (similarly for terms and formulas).

Example 1. Let $\Sigma_{\mathcal{A}}$ be a signature that contains a binary function read, a ternary function write, no constants, and no predicate symbols. The theory $T_{\mathcal{A}}$ of arrays (without extensionality) is defined by the following formulas [21]:

$$\forall a\ \forall i\ \forall e\ [\mathsf{read}(\mathsf{write}(a, i, e), i) = e]$$
$$\forall a\ \forall i\ \forall j\ \forall e\ [i \neq j \Rightarrow \mathsf{read}(\mathsf{write}(a, i, e), j) = \mathsf{read}(a, j)].$$

A formula F is T-satisfiable or T-consistent if $F \wedge T$ is satisfiable in the first-order sense (*i.e.*, there is an interpretation that satisfies it). A formula F is called T-unsatisfiable or T-inconsistent if it is not T-satisfiable. For formulas F and G, F T-entails G (in symbols $F \models_T G$) if $F \wedge \neg G$ is T-inconsistent.

Definition 1. *Let $\Sigma_{\mathcal{Z}}$ be a signature that contains the constant symbols $\{0, \pm 1, \pm 2, \ldots\}$, a binary function symbol $+$, a unary function symbol $-$, and a binary predicate symbol $\leq$. The theory of Linear Integer Arithmetic, which we will denote by $\mathcal{Z}$, is the $\Sigma_{\mathcal{Z}}$-theory defined by the set of closed $\Sigma_{\mathcal{Z}}$-formulas that are true in the standard model (an interpretation whose domain is $\mathbb{Z}$, in which the symbols in $\Sigma_{\mathcal{Z}}$ are interpreted according to their standard meaning over $\mathcal{Z}$).*

We will use relation symbols like $<$ that do not appear in $\Sigma_{\mathcal{Z}}$, and also multiplication by a constant (which is to be interpreted as repeated addition); these are only syntactic shorthands. We will frequently view an integer assignment A as the set of formulas $\{v = A(v) \mid v \in \mathcal{V}\}$, where $A(v)$ is viewed as a $\Sigma_{\mathcal{Z}}$-term. An integer assignment A viewed as a set of formulas is always $\mathcal{Z}$-consistent. If A is an integer assignment and A satisfies an integer linear formula C, it is also the case that $A \models_{\mathcal{Z}} C$. If A is an integer assignment, T is a theory and F is a formula, we will say that A is a T-model of F if A is T-consistent and $A \models_{\mathcal{Z} \cup T} F$. Note that a T-model is not a first-order model.

A *Σ-interface atom* is a Σ-atomic formula (*i.e.*, the application of a predicate symbol or equality), possibly annotated with a variable symbol, *e.g.*, $(x = y)^v$. The meaning of a Σ-interface atom with no annotation remains the same. An annotated Σ-interface atom ϕ^v denotes $\phi \Leftrightarrow v > 0$. A set of Σ-interface atoms will often be used to denote their conjunction.

Definition 2 (ILP Modulo T Instance). *An* ILP Modulo (Theory) T instance*, where the signature Σ of T is disjoint from $\Sigma_{\mathcal{Z}}$, is a triple of the form C, I, O, where C is an integer linear formula, I is a set of Σ-interface atoms, and O is an objective function. The variables that appear in both C and I are called interface variables.*

An ILP Modulo T instance can be thought of as an integer linear program that contains terms which have meaning in T. In Definition 2, the interface atoms (elements of I) are separated from the linear constraints, *i.e.*, there are no Σ-terms embedded within integer linear constraints. This is not a restriction, because every set of $(\Sigma \cup \Sigma_{\mathcal{Z}})$-atomic formulas can be written in separate form [19, "Variable Abstraction"].

Example 2. Let Σ be a signature that contains the unary function symbol f. The formula $f(x+1) + f(y+2) \geq 3$ (where x and y are variable symbols) can be written in separate form as $C = \{v_3 + v_4 \geq 3, v_1 = x + 1, v_2 = y + 2\}$ and $I = \{v_3 = f(v_1), v_4 = f(v_2)\}$. C is an integer linear formula; I is a set of Σ-interface atoms; and Σ is disjoint from $\Sigma_{\mathcal{Z}}$. Variable abstraction introduced new variables, $v_1, \ldots, v_4$. C and I only share variable symbols.

Let A be the assignment $\{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 0\}$. Clearly $A \models_{\mathcal{Z}} C$. However, $A \not\models_{\emptyset} I$, where $\emptyset$ stands for the *theory of uninterpreted functions* (also called the empty theory, because it has an empty set of formulas). The reason is that $v_1 = v_2$ but $f(v_1) \neq f(v_2)$. In contrast, the assignment $A' = \{x = 2, y = 1, v_1 = 3, v_2 = 3, v_3 = 3, v_4 = 3\}$ is a $\emptyset$-model of $C \wedge I$ per our definition, as A' is $\emptyset$-consistent and $A' \models_{\mathcal{Z} \cup \emptyset} C \wedge I$.

2.2 Transition System

Definition 3 (Difference Constraint). *A difference constraint is a constraint of the form $v_i \leq c$ or $v_i - v_j \leq c$, where v_i and v_j are integer variables and c is an integer constant.*

Definition 4 (Subproblem). *A subproblem is a pair of the form $\langle C, D \rangle$, where C is a set of constraints and D is a set of difference constraints.*

In a subproblem $\langle C, D \rangle$, we distinguish between the arbitrary constraints in C and the simpler constraints in D in order to provide a good interface for the interaction between the core ILP solver and background theory solvers that only understand difference logic, *i.e.*, a limited fragment of $\mathcal{Z}$. It is the responsibility of the core solver to notify the theory solver about the difference constraints that hold. Difference constraints are clearly a special case of integer linear constraints.

Definition 5 (State). *A state of* BC(T) *is a tuple $P \parallel A$, where P is a set of subproblems, and A is either the constant* `None`*, or an assignment. If A is an assignment, it can optionally be annotated with the superscript $-\infty$.*

Our abstract framework maintains a list of open subproblems, because it is designed to allow different branching strategies. This is in contrast to an algorithm

like CDCL that does not keep track of subproblems explicitly. There, subproblems are implicit, *i.e.*, backtracking can reconstruct them. ILP solvers branch over non-Boolean variables in arbitrary ways, thus mandating that we explicitly record subproblems.

In a state $P \parallel A$, the assignment A is the best known (T-consistent) solution so far, if any. It has a superscript $-\infty$ if it satisfies all the constraints, but is not optimal because the IMT instance admits solutions with arbitrarily low objective values. If this is the case, it is useful to provide an assignment and to also report that no optimal assignment exists.

The interface atoms I and the objective function O are not part of the BC(T) states because they do not change over time. $\mathsf{obj}(A)$ denotes the value of the objective function O under assignment A: if $O = \sum_i c_i v_i$, then $\mathsf{obj}(A) = \sum_i c_i \cdot A(v_i)$. The objective function itself is not an argument to obj because it will be clear from the context which objective function we are referring to. For convenience, we define $\mathsf{obj}(\texttt{None}) = +\infty$ and $\mathsf{obj}(A^{-\infty}) = -\infty$. Function $\mathsf{lb}(\langle C, D\rangle)$ returns a lower bound for the possible values of the objective function O for the subproblem $\langle C, D\rangle$: by definition, there is no A such that A satisfies $C \wedge D$ and $\mathsf{obj}(A) < \mathsf{lb}(\langle C, D\rangle)$.

Figure 1 defines the *transition relation* $\longrightarrow$ of BC(T) (a binary relation over states). In the rules, c and d always denote integer linear constraints and difference constraints. C (possibly subscripted) denotes an integer linear formula (set of integer linear constraints), while D denotes a set of difference constraints. $C\ c$ stands for the set union $C \cup \{c\}$, under the implicit assumption that $c \notin C$; similarly for $D\ d$. C and D are always well-formed sets, *i.e.*, they contain no syntactically duplicate elements. P and P' stand for sets of syntactically distinct subproblems, while A and A' are integer assignments. $P \uplus Q$ denotes the union $P \cup Q$, under the implicit assumption that the two sets are disjoint. The intuitive meaning of the different BC(T) rules is the following:

`Branch`
: Case-split on a subproblem $\langle C, D\rangle$, by replacing it with two or more different subproblems $\langle C_i, D\rangle$. If there is a satisfying assignment for $C \wedge D$, this assignment will also satisfy $C_i \wedge D$ for some i, and conversely.

`Learn`, T-`Learn`, `Propagate`
: Add an entailed constraint (in the case of `Learn` and T-`Learn`) or difference constraint (`Propagate`) to a subproblem. T-`Learn` takes the theory T into account. T-`Learn` is strictly more powerful than `Learn`. We retain the latter as a way to denote transitions that do not involve theory reasoning.

`Forget`
: Remove a constraint entailed by the remaining constraints of a subproblem.

`Drop`, `Prune`
: Eliminate a subproblem either because it is unsatisfiable (`Drop`), or because it cannot lead to a solution better than the one already known.

$$\texttt{Branch}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C_i, D\rangle \mid 1 \le i \le n\} \parallel A \quad \text{if} \begin{cases} n > 1 \\ D \models_{\mathcal{Z}} (C \Leftrightarrow \bigvee_{1 \le i \le n} C_i) \\ C_i \text{ are syntactically distinct} \end{cases}$$

$$\texttt{Learn}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C\ c, D\rangle\} \parallel A \quad \text{if } C \wedge D \models_{\mathcal{Z}} c$$

$$T\text{-}\texttt{Learn}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C\ c, D\rangle\} \parallel A \quad \text{if } C \wedge D \wedge I \models_{\mathcal{Z} \cup T} c$$

$$\texttt{Propagate}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C, D\ d\rangle\} \parallel A \quad \text{if } C \wedge D \models_{\mathcal{Z}} d$$

$$\texttt{Forget}\quad P \uplus \{\langle C\ c, D\rangle\} \parallel A \longrightarrow P \cup \{\langle C, D\rangle\} \parallel A \quad \text{if } C \wedge D \models_{\mathcal{Z}} c$$

$$\texttt{Drop}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A \quad \text{if } C \wedge D \text{ is integer-inconsistent}$$

$$\texttt{Prune}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A \quad \text{if} \begin{cases} A \neq \texttt{None} \\ \mathsf{lb}(\langle C, D\rangle) \ge \mathsf{obj}(A) \end{cases}$$

$$\texttt{Retire}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow P \parallel A' \quad \text{if} \begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ \mathsf{obj}(A') < \mathsf{obj}(A) \\ \text{for any } T\text{-model } B \text{ of } C \wedge D \wedge I,\ \mathsf{obj}(A') \le \mathsf{obj}(B) \end{cases}$$

$$\texttt{Unbounded}\quad P \uplus \{\langle C, D\rangle\} \parallel A \longrightarrow \emptyset \parallel A'^{\,-\infty} \quad \text{if} \begin{cases} A' \text{ is a } T\text{-model of } C \wedge D \wedge I \\ \mathsf{obj}(A') \le \mathsf{obj}(A) \\ \text{for any integer } k, \text{ there exists a } T\text{-model } B \text{ of } C \wedge D \wedge I \\ \quad \text{such that } \mathsf{obj}(B) < k \end{cases}$$

Fig. 1. The BC(T) Transition System

`Retire`, `Unbounded`
: The solution to a subproblem becomes the new incumbent solution, as long as it improves upon the objective value of the previous solution. If there are solutions with arbitrarily low objective values, we don't need to consider other subproblems.

The observant reader will have noticed that the T-`Learn` rule is very powerful, *i.e.*, it allows for combined $\mathcal{Z} \cup T$-entailment. This is in pursuit of generality. Our completeness strategy (Theorem 3) will not depend in any way on performing combined arithmetic and theory reasoning, but only on extracting equalities and disequalities from the difference constraints. Entailment modulo $\mathcal{Z} \cup T$ is required if we want to learn clauses, because they are represented as linear constraints. Interesting implementations of BC(T) may go beyond clauses and apply T-`Learn` for theory-specific cuts.

We define the binary relations $\longrightarrow^+$ and $\longrightarrow^*$ over BC(T) states as follows: $S \longrightarrow^+ S'$ if $S \longrightarrow S'$, or there exists some state Q such that $S \longrightarrow^+ Q$ and $Q \longrightarrow S'$. $S \longrightarrow^* S'$ if $S = S'$ or $S \longrightarrow^+ S'$. When convenient, we will annotate a transition arrow between two BC(T) states with the name of the rule that relates them, for example $S \underset{\texttt{Branch}}{\longrightarrow} S'$.

A *starting state* for BC(T) is a state of the form $\{\langle C, \emptyset \rangle\} \parallel \texttt{None}$, where C is the set of integer linear constraints of an ILP Modulo T instance. A *final state* is a state of the form $\emptyset \parallel A$ (A can also be `None`, or an assignment annotated with $-\infty$).

2.3 Soundness and Completeness

Throughout this Section, we assume an IMT instance with objective function O and a set of interface atoms I. Theorems 1 and 2 characterize BC(T) soundness. A version of this paper with proofs is available through arXiv [20].

Theorem 1. *For a formula C, if $\{\langle C, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^* \emptyset \parallel \texttt{None}$, then $C \wedge I$ is $\mathcal{Z} \cup T$-unsatisfiable.*

Theorem 2. *For a formula C and an assignment A, if*

$$\{\langle C, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^* \emptyset \parallel A$$

where $A \neq \texttt{None}$, then (a) A is a T-model of $C \wedge I$, and (b) there is no assignment B such that B is a T-model of $C \wedge I$ and $\mathsf{obj}(B) < \mathsf{obj}(A)$.

Definition 6 (Stably-Infinite Theory). *A Σ-theory T is called stably-infinite if for every T-satisfiable quantifier-free Σ-formula F there exists an interpretation satisfying $F \wedge T$ whose domain is infinite.*

Definition 7 (Arrangement). *Let E be an equivalence relation over a set of variables V. The set*

$$\alpha(V, E) = \{x = y \mid xEy\} \cup \{x \neq y \mid x, y \in V \text{ and not } xEy\}$$

is the arrangement of V induced by E.

Note that $\mathcal{Z}$ is a stably-infinite theory. We build upon the following result on the combination of signature-disjoint stably-infinite theories:

Fact 1 (Combination of Stably-Infinite Theories [23,31,19]) *Let T_i be a stably-infinite Σ_i-theory, for $i = 1, 2$, and let $\Sigma_1 \cap \Sigma_2 = \emptyset$. Also, let Γ_i be a conjunction of Σ_i literals. $\Gamma_1 \cup \Gamma_2$ is $(T_1 \cup T_2)$-satisfiable iff there exists an equivalence relation E of the variables shared by Γ_1 and Γ_2 such that $\Gamma_i \cup \alpha(V, E)$ is T_i-satisfiable, for $i = 1, 2$.*

Decidability for the combination of $T_1 = \mathcal{Z}$ and another stably-infinite theory follows trivially, as we can pick an arrangement over the variables shared by the two sets of literals non-deterministically and perform two T_i-satisfiability checks. We show that BC(T) can be applied in a complete way by meeting the hypotheses of Fact 1.

Theorem 3 (Completeness). BC(T) *provides a complete optimization procedure for the ILP Modulo T problem, where T is a decidable stably-infinite theory.*

Proof (Sketch). Let $\mathcal{C}, I, O$ be an ILP Modulo T instance. Assume that

$$\{\langle \mathcal{C}, \emptyset \rangle\} \parallel \texttt{None} \longrightarrow^* P \parallel A,$$

and that for every $\langle C, D \rangle \in P$ the following conditions hold: (a) there is an equivalence relation E_D over the set of interface variables V of the ILP Modulo T instance, such that $D \models_{\mathcal{Z}} \alpha(V, E_D)$, and (b) either $D \models_{\mathcal{Z}} v > 0$ or $D \models_{\mathcal{Z}} v \leq 0$ for every v that appears as the annotation of an interface atom in I. Then we can solve the IMT instance to optimality as follows. For every subproblem $\langle C, D \rangle \in P$, $C \wedge D \wedge I$ $\mathcal{Z} \cup T$-entails the following set of literals:

$$\begin{aligned} \{\phi \mid \phi \in I \text{ and } \phi \text{ is not annotated}\} \cup \\ \{\phi \mid \phi^v \in I \text{ and } D \models_{\mathcal{Z}} v > 0\} \cup \\ \{\neg\phi \mid \phi^v \in I \text{ and } D \models_{\mathcal{Z}} v \leq 0\} \cup \\ \alpha(V, E_D) \end{aligned}$$

If the set of literals is T-unsatisfiable, then $C \wedge D \wedge I$ is $\mathcal{Z} \cup T$-unsatisfiable. If it is T-satisfiable, any integer solution for $C \wedge D$ will be a T-model. For the T-unsatisfiable subproblems, we apply T-`Learn` to learn an integer-infeasible constraint (*e.g.*, $0 < 0$) and subsequently apply `Drop`. If all the subproblems are T-unsatisfiable, we reach a final state $\emptyset \parallel A$. If there are T-satisfiable subproblems, it suffices to let a (complete) branch-and-cut ILP algorithm run to optimality, as we have already established T-consistency. The basic steps of such algorithms can be described by means of BC(T) steps. Note that unbounded objective functions do not hinder completeness: it suffices to recognize an unbounded subproblem [4] and apply `Unbounded`.

A systematic branching strategy can guarantee that after a finite number of steps, the difference constraints of every subproblem entail an arrangement. For every pair of interface variables x and y and every subproblem, we apply the `Branch` rule to obtain three new subproblems, each of which contains one of the constraints $x - y < 0$, $x - y = 0$, and $x - y > 0$. The `Propagate` rule then applies to all three subproblems. Similarly, we branch to obtain a truth value for $v > 0$ for every v that appears as the annotation of an interface atom.

3 SMT as IMT

In Section 2, we provided a sound and complete optimization procedure for the combination of ILP and a stably-infinite theory (Theorems 1, 2, and 3). We will now demonstrate how to deal with propositional structure, so that we can use this procedure for SAT Modulo $\mathcal{Z} \cup T$ problems, where T is stably-infinite. In essence, our goal is to flatten propositional structure into linear constraints.

3.1 Bounding $\mathcal{Z} \cup T$ Instances

As a prerequisite for dealing with propositional structure, we show how to bound integer terms in quantifier-free formulas while preserving $\mathcal{Z} \cup T$-satisfiability. We build upon well-known results for ILP [5]. Similar ideas have been applied to $\mathcal{Z}$ [30]. Our results go beyond the bounds for $\mathcal{Z}$, in that we take into account background theories and objective functions.

We will say that a term is Σ-rooted if (at its root) it is an application of a function symbol from the signature Σ. Let Σ_0 and Σ_1 be signatures such that $\Sigma_0 \cap \Sigma_1 = \emptyset$. Given a $\Sigma_0 \cup \Sigma_1$-formula F, we will refer to the Σ_i-rooted terms that appear directly under predicate and function symbols from Σ_{1-i} as the Σ_i-interface terms in F. Interface terms are the ones for which variable abstraction (Example 2) introduces fresh variables.

Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$, and F be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula. We denote by $\mathsf{intf}_{\mathcal{Z}}(F)$ and $\mathsf{intf}_{\Sigma}(F)$ the sets of $\Sigma_{\mathcal{Z}}$-interface terms and Σ-interface terms in F, and by $\mathsf{intf}(F)$ the union $\mathsf{intf}_{\Sigma}(F) \cup \mathsf{intf}_{\mathcal{Z}}(F)$. Let $\mathsf{atoms}_{\mathcal{Z}}(F)$ be the set of atomic formulas in F that are applications of $\leq$; without loss of generality, we will assume that formulas contain no arithmetic equalities or other kinds of inequalities. Also, let $\mathsf{maxc}(F)$ be the maximum absolute value among integer coefficients in F plus one, and $\mathsf{vars}_{\mathcal{Z}}(F)$ be the set of variable symbols that appear directly under predicate and function symbols from $\Sigma_{\mathcal{Z}}$. By $o(M)$ we denote the interpretation of linear expression o under the first-order model M. We finally define $\mathsf{bounds}(F, \rho) = \{-\rho \leq t \wedge t \leq \rho \mid t \in \mathsf{vars}_{\mathcal{Z}}(F) \cup \mathsf{intf}(F)\}$, for positive integers ρ.

Theorem 4. *Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$ and T be a stably-infinite Σ-theory. Let F be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula and o an objective function. Let $k = |\mathsf{atoms}_{\mathcal{Z}}(F)| + |\mathsf{intf}(F)| + |\mathsf{vars}_{\mathcal{Z}}(F)| - 1$, $m = |\mathsf{intf}_{\mathcal{Z}}(F)| + k$, and $n = |\mathsf{intf}(F)| + |\mathsf{vars}_{\mathcal{Z}}(F)|$. Finally, let*

$$\rho \;=\; (2n+k)^3[(m+2)\,\mathsf{maxc}(F)]^{4m+12}.$$

*If there is a first-order model M such that $M \models F \wedge \mathcal{Z} \wedge T$ and M is a finite optimum for F with respect to o (*i.e.*, there is some integer constant c such that $M \models o = c$ and there is no model M' such that $M' \models F \wedge \mathcal{Z} \wedge T$ and $M' \models o < c$), then $\{F\} \cup \mathsf{bounds}(F, \rho) \cup \{o = o(M)\}$ is $\mathcal{Z} \cup T$-satisfiable.*

We provide a proof through arXiv [20]. Intuitively, given a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula F and an objective function o, Theorem 4 allows us to bound the integer terms of F while preserving (finite) optimality. [1]

3.2 Propositional Structure

Let Σ be a signature such that $\Sigma_{\mathcal{Z}} \cap \Sigma = \emptyset$ and T be a stably-infinite Σ-theory. Throughout this Section, F will be a quantifier-free $\Sigma_{\mathcal{Z}} \cup \Sigma$-formula and O will be an objective function. We show how to encode F as the conjunction of a set C of integer linear constraints and a set I of Σ-interface atoms, while preserving optimality with respect to O. We apply a Tseitin-like algorithm, *i.e.*, we recursively introduce $\{0,1\}$-constrained variables for subformulas of F.

The most interesting part is dealing with predicate symbols from Σ and $\Sigma_{\mathcal{Z}}$. For the former we simply introduce annotated Σ-interface atoms, *e.g.*, $[p(x)]^v$. For $\Sigma_{\mathcal{Z}}$, we can assume that we are only confronted with inequalities of the form $\phi = (\sum_i \ c_i \cdot v_i \leq r)$, because other relations can be expressed in terms of $\leq$ and the propositional connectives. Also, we only have to deal with sums over variable symbols, because variable abstraction takes care of terms that involve Σ. We define a variable $v(\phi)$ such that $v(\phi) \Leftrightarrow \phi$ as follows. By bounding all variables as per Theorem 4, we compute m and k such that $m < \sum_i c_i \cdot v_i \leq k$ always holds. The direction $v(\phi) \Rightarrow \phi$ can be expressed as $\sum_i c_i \cdot v_i \leq r + (k - r) \cdot (1 - v(\phi))$; for the opposite direction we have $\sum_i c_i \cdot v_i > r + (m - r) \cdot v(\phi)$.

With atomic formulas taken care of, what remains is propositional connectives; we encode them by using clauses in the standard fashion. Clauses appear as part of our collection of ILP constraints: $\vee_i l_i$ is equivalent to $\sum_i l_i \geq 1$. (For translating a clause to a linear expression, a negative literal $\neg v_i$ appears as $1 - v_i$ while a positive literal remains intact.)

Note that the (possibly astronomical) coefficients we compute only serve the purpose of representing formulas as sets of linear constraints. Their magnitude does not necessarily have algorithmic side-effects. In the worst case, the initial continuous relaxation will be weak, but relaxations will become stronger once we start branching on the Boolean variables. This is no worse than Lazy SMT, where linear constraints are only applicable once the SAT core assigns the corresponding Boolean variables.

4 Implementation and Experiments

IMT first appeared in the context of architectural synthesis for aerospace systems [16]. Our approach combined an ILP solver with a custom decision procedure for real-time constraints. We implemented the combination in the CoBaSA tool. The CoBaSA manifestation of IMT predates BC(T). More recently, we implemented a BC(T)-based solver, which we call Inez.

[1] A solver that relies on Theorem 4 for bounding can detect unboundedness by imposing the additional constraint $o < o(M)$, re-computing bounds, and solving the resulting instance. If the updated instance is satisfiable, the original is unbounded.

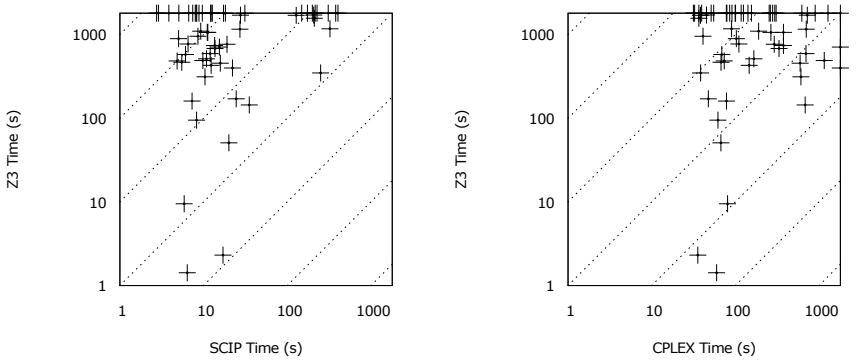


Fig. 2. Z3 versus SCIP and CPLEX (Synthesis Instances)

Our experimental evaluation is twofold. First, we show that an ILP core is essential for the practicality of our synthesis approach. This part of the evaluation does not deal with BC(T) in any way, but it nevertheless provides evidence that IMT enables new applications. Second, we compare our BC(T) prototype against Z3 [8] and MathSAT [14] using benchmarks from the SMT-LIB.

4.1 Motivation

In the past, we applied CoBaSA to solve the architectural synthesis problem for the real, production data from the 787, which was provided to us by Boeing [16]. We have made available a family of 60 benchmark instances derived from Boeing problems, 47 of which are unsatisfiable. [2] We will use these instances to evaluate the suitability of SMT and ILP solvers as the core of a combination framework for synthesis, which is a key application area for IMT.

We briefly describe the synthesis problem that gives rise to our benchmarks. The basic components for this problem are cabinets (providing resources like CPU time, bandwidth, battery backup, and memory), software applications (that consume resources), and global memory spaces (that also consume resources). Applications and memories have to be mapped to cabinets subject to various constraints, *e.g.*, resource allocation and fault tolerance. Applications communicate via a publish-subscribe network. Messages are aggregated into virtual links that are multicast. The network and messages are subject to various constraints, *e.g.*, bandwidth and scheduling constraints. The instances differ in the numbers of different components, the amounts of different resources, and the collection of structural and scheduling requirements they encode.

The instances are $\{0, 1\}$-ILP (also known as Pseudo-Boolean). There are multiple ways to encode $\{0, 1\}$-ILP problems as SMT-LIB instances. A direct translation led to SMT problems that Z3 could not solve, so we tried several encodings, most of which yielded similar results. One encoding was significantly better than the rest, and it works as follows. Some of the linear constraints are clauses, *i.e.*,

[2] `http://www.ccs.neu.edu/home/vpap/benchmarks.html`

of the form $\sum l_i \geq 1$ for literals l_i. It makes sense to help SMT solvers by encoding such constraints as disjunctions of literals instead of inequalities. To do this, we declare all variables to be Boolean. Since almost all variables also appear in arithmetic contexts where they are multiplied by constants greater than 1, we translate such constraints as demonstrated by the following example: the linear constraint $x + y + 2z \geq 2$ becomes `(>= (+ (ite x 1 0) (ite y 1 0) (ite z 2 0)) 2)`.

Figure 2 visualizes the behavior of Z3 versus SCIP and CPLEX. SCIP solves all instances, while CPLEX solves all but 3. Z3 solves 5 out of 13 satisfiable and 30 out of 47 unsatisfiable instances. Strictly speaking, the only theory involved is $\mathcal{Z}$. However, the instances do contain collections of scheduling theory lemmas [16] recorded by CoBaSA in the process of solving synthesis problems. Therefore, our setup simulates the kinds of queries a core solver would be confronted with, when coupled with our scheduling solver. With suitability for synthesis as the evaluation criterion, this is the most rigorous comparison we can perform without implementing and optimizing the combination of SMT with scheduling. Both ILP solvers significantly outperform Z3, demonstrating the potential of a general ILP-based combination framework.

4.2 BC(T) Implementation

Inez is implemented as an unobtrusive extension of SCIP. Namely, we have extended SCIP with a congruence closure procedure (*constraint handler* in SCIP terms), and also provide an SMT-LIB frontend. The overall architecture of SCIP matches BC(T). Subproblems (called *nodes*) are created by branching (`Branch`) and eliminated by operations semantically very similar to `Drop`, `Prune`, `Retire`, and `Unbounded`. SCIP employs various techniques for cut generation (`Learn`).

Like most modern MIP solvers, SCIP relies heavily on linear relaxations. While not explicitly mentioned in BC(T), linear relaxations fit nicely: (a) lb relies on continuous relaxations, as the best integral solution can be at most as good as the best non-integral solution. (b) Solutions to relaxations frequently guide branching, *e.g.*, if a solution assigns a non-integer value r to variable v, it makes sense to branch around r ($v \geq \lceil r \rceil$ or $v \leq \lfloor r \rfloor$). (c) If some relaxation is infeasible, then the corresponding subproblem is infeasible and `Drop` applies, while (d) `Retire` or `Unbounded` applies to T-consistent integer solutions.

BC(T) proposes difference constraints as a channel of communication with theory solvers (`Propagate` rule). Inez implements `Propagate` as follows. For every pair of variables x and y whose (dis)equality is of interest to the theory solver, Inez introduces a variable $d_{x,y}$ and imposes the constraint $d_{x,y} = x - y$. When SCIP fixes the lower bound of $d_{x,y}$ to l, the theory solver is notified of the difference constraint $l \leq x - y$ (similarly for the upper bound). We generally need quadratically many such auxiliary variables. This is not necessarily a practical issue, because most pairs of variables are irrelevant.

Our congruence closure procedure takes offsets into account [25]. In addition to standard propagation based on congruence closure, Inez applies techniques specific to the integer domain. Notably, if x is bounded between a and b, and

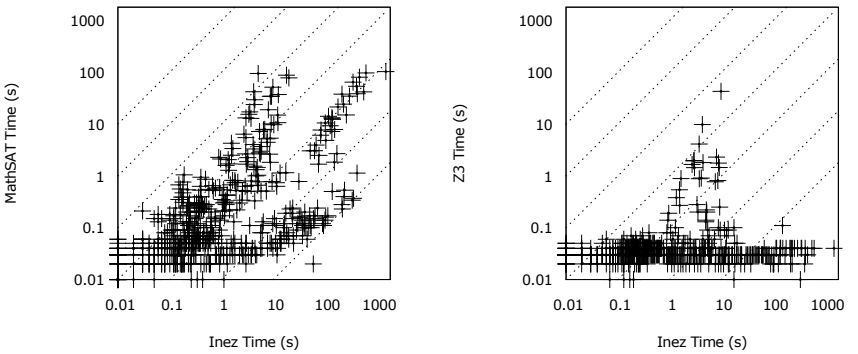


Fig. 3. Inez versus Z3 and MathSAT (SMT-LIB Instances)

for every value of k in $[a, b]$, $f(k)$ is bounded between l and u, it follows that $l \leq f(x) \leq u$, *i.e.*, we can impose bounds on $f(x)$. a and b do not have to be constants, *e.g.*, it may be the case that $m \leq d_{x,y} = x - y \leq n$ and $f(y + m), \ldots, f(y + n)$ are bounded. We apply this idea dynamically (to benefit from local bounds) and not just as preprocessing.

BC(T) does not preclude techniques that target special classes of linear constraints. For example, an implementation can use the two-watched-literal scheme to accelerate Boolean Constraint Propagation on clauses. SCIP implements such techniques. Note that IMT does not strive to replace propositional reasoning, but rather to shift a broader class of constraints to the core solver.

4.3 BC(T) on SMT-LIB

We experimentally evaluate Inez against MathSAT and Z3, based on the most relevant SMT-LIB category, which is `QF_UFLIA` (Quantifier-Free Linear Integer Arithmetic with Uninterpreted Functions). Z3 and MathSAT solve all 562 benchmarks, and so does Inez. While Inez is generally slower than the more mature SMT solvers, the majority of the benchmarks (338) require less than a second, 462 benchmarks require less than 10 seconds, and 528 less than 100 seconds. The integer-specific kind of propagation outlined in Section 4.2 is crucial; we only solve 490 instances with this technique disabled. Figure 3 visualizes our experiments.

Interestingly, the underlying SCIP solver learns no cutting planes whatsoever for 362 out of the 562 instances. For the remaining instances the number of cuts is limited. Namely, 126 instances lead to a single cut, 61 lead to 2 cuts, and the remaining 13 instances lead to 9 cuts or less. Based on this observation, the branching part of Inez's branch-and-cut algorithm is being stressed here. We have not yet tried to optimize branching heuristics, so there is plenty of room for improvement. More importantly, the instances are not representative of arithmetic-heavy optimization problems, where we would expect more cuts.

A final observation is that SCIP performs floating-point (FP) arithmetic, which may lead to wrong answers. Interestingly, Inez provides no wrong answers

for the benchmark set in question, *i.e.*, the instances do not pose numerical difficulties. The fact that we learn very few cutting planes partially explains why. There is little room for learning anything at all, let alone for learning something unsound.

5 Related Work

Branch-and-Cut: Branch-and-cut algorithms [22] combine branch-and-bound with cutting plane techniques, *i.e.*, adding violated inequalities (cuts) to the linear formulation. Different cut generation methods have been studied for general integer programming problems, starting with the seminal work of Gomory [13]. Cuts can also be generated in a problem-specific way, *e.g.*, for TSP [15]. Problem-specific cuts are analogous to theory lemmas in IMT.

Nelson-Oppen: The seminal work of Nelson and Oppen [23] provided the foundations for combining decision procedures. Tinelli and Harandi [31] revisit the Nelson-Oppen method and propose a non-deterministic variant for non-convex stably-infinite theories. Manna and Zarba provide a detailed survey of Nelson-Oppen and related methods [19].

SMT: ILP Modulo Theories resembles Satisfiability Modulo Theories, with ILP as the core formalism instead of SAT. SMT has been the subject of active research over the last decade [3,10,27,8]. Nieuwenhuis, Oliveras and Tinelli [27] present the abstract DPLL(T) framework for reasoning about lazy SMT. Different fragments of Linear Arithmetic have been studied as background theories for SMT [11,14]. Extensions of SMT support optimization [26,6,29].

Generalized CDCL: A family of solvers that generalize CDCL-style search to richer logics recently emerged [17,28,18,9]. This research direction can be viewed as progress towards SMT with a non-propositional core. Our work is complementary, in the sense that we do not focus on the core solver, but rather provide a way to combine a non-Boolean core with theories.

Inexact Solvers: Linear and integer programming solvers generally perform FP (and thus inexact) calculations. Faure et al. experiment with the inexact CPLEX solver as a theory solver [12] and observe wrong answers. For many applications, numerical inaccuracies are not a concern, *e.g.*, the noise in the model overshadows the floating point error intervals. However, accuracy is often critical. Recent work [24,7] proposes using FP arithmetic as much as possible (especially for solving continuous relaxations) while preserving safety. IMT solvers can be built on top of both exact and inexact solvers.

6 Conclusions and Future Work

We introduced the ILP Modulo Theories (IMT) framework for describing problems that consist of linear constraints along with background theory constraints.

We did this via the BC(T) transition system that captures the essence of branch-and-cut for solving IMT problems. We showed that BC(T) is a sound and complete optimization procedure for the combination of ILP with stably-infinite theories. We conducted a detailed comparison between SMT and IMT.

Many interesting research directions now open up. We could try to relax requirements on the background theory (*e.g.*, stably-infiniteness, signature disjointness) while preserving soundness and completeness. We anticipate interesting connections between IMT and other paradigms, *e.g.*, SMT, constraint programming, cut generation, and decomposition. Additionally, the BC(T) architecture seems to allow for significant parallelization. Finally, we believe that IMT has the potential to enable interesting new applications.

Acknowledgements. We would like to thank Harsh Raju Chamarthi, Mitesh Jain, and the anonymous reviewers for their valuable comments and suggestions.

References

1. CPLEX, `http://www-01.ibm.com/software/integration/optimization/cplex-optimizer/`
2. Achterberg, T.: Constraint Integer Programming. PhD thesis, Technische Universitat Berlin (2007)
3. Barrett, C.W., Dill, D.L., Stump, A.: Checking Satisfiability of First-Order Formulas by Incremental Translation to SAT. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 236–249. Springer, Heidelberg (2002)
4. Byrd, R.H., Goldman, A.J., Heller, M.: Recognizing Unbounded Integer Programs. Operations Research 35(1), 140–142 (1987)
5. Papadimitriou, C., Steiglitz, K.: Combinatorial Optimization: Algorithms and Complexity, 2nd edn. Dover (1998)
6. Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R., Stenico, C.: Satisfiability Modulo the Theory of Costs: Foundations and Applications. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 99–113. Springer, Heidelberg (2010)
7. Cook, W., Koch, T., Steffy, D.E., Wolter, K.: An Exact Rational Mixed-Integer Programming Solver. In: Günlük, O., Woeginger, G.J. (eds.) IPCO 2011. LNCS, vol. 6655, pp. 104–116. Springer, Heidelberg (2011)
8. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
9. de Moura, L., Jovanović, D.: A Model-Constructing Satisfiability Calculus. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 1–12. Springer, Heidelberg (2013)
10. de Moura, L., Ruess, H.: Lemmas on Demand for Satisfiability Solvers. In: SAT (2002)
11. Dutertre, B., de Moura, L.: A Fast Linear-Arithmetic Solver for DPLL(T). In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 81–94. Springer, Heidelberg (2006)

12. Faure, G., Nieuwenhuis, R., Oliveras, A., Rodríguez-Carbonell, E.: SAT Modulo the Theory of Linear Arithmetic: Exact, Inexact and Commercial Solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 77–90. Springer, Heidelberg (2008)
13. Gomory, R.E.: Outline of an algorithm for integer solutions to linear programs. Bulletin of the AMS 64, 275–278 (1958)
14. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8, 1–27 (2012)
15. Grotschel, M., Holland, O.: Solution of large-scale symmetric travelling salesman problems. Mathematical Programming 51, 141–202 (1991)
16. Hang, C., Manolios, P., Papavasileiou, V.: Synthesizing Cyber-Physical Architectural Models with Real-Time Constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 441–456. Springer, Heidelberg (2011)
17. Jovanović, D., de Moura, L.: Cutting to the Chase: Solving Linear Integer Arithmetic. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 338–353. Springer, Heidelberg (2011)
18. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)
19. Manna, Z., Zarba, C.: Combining Decision Procedures. In: 10th Anniversary Colloquium of UNU/IIST (2002)
20. Manolios, P., Papavasileiou, V.: ILP Modulo Theories. CoRR, abs/1210.3761 (2012)
21. McCarthy, J.: Towards a Mathematical Science of Computation. In: Congress IFIP-1962 (1962)
22. Mitchell, J.E.: Branch-and-Cut Algorithms for Combinatorial Optimization Problems. In: Handbook of Applied Optimization, pp. 223–233. Oxford University Press (2000)
23. Nelson, G., Oppen, D.C.: Simplification by Cooperating Decision Procedures. TOPLAS 1, 245–257 (1979)
24. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer linear programming. Mathematical Programming 99, 283–296 (2004)
25. Nieuwenhuis, R., Oliveras, A.: Congruence Closure with Integer Offsets. In: Vardi, M.Y., Voronkov, A. (eds.) LPAR 2003. LNCS, vol. 2850, Springer, Heidelberg (2003)
26. Nieuwenhuis, R., Oliveras, A.: On SAT Modulo Theories and Optimization Problems. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 156–169. Springer, Heidelberg (2006)
27. Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL(T). JACM 53(6), 937–977 (2006)
28. Cotton, S.: Natural domain SMT: A preliminary assessment. In: Chatterjee, K., Henzinger, T.A. (eds.) FORMATS 2010. LNCS, vol. 6246, pp. 77–91. Springer, Heidelberg (2010)
29. Sebastiani, R., Tomasi, S.: Optimization in SMT with LA(Q) Cost Functions. In: IJCAR (2012)
30. Seshia, S.A., Bryant, R.E.: Deciding Quantifier-Free Presburger Formulas Using Parameterized Solution Bounds. In: LICS (2004)
31. Tinelli, C., Harandi, M.: A New Correctness Proof of the Nelson-Oppen Combination Procedure. In: FroCoS (1996)

Smten: Automatic Translation of High-Level Symbolic Computations into SMT Queries*

Richard Uhler[1] and Nirav Dave[2]

[1] Massachusetts Institute of Technology,
Computer Science and Artificial Intelligence Laboratory, Cambridge, MA, USA
ruhler@csail.mit.edu
[2] SRI International, Computer Science Laboratory, Menlo Park, CA, USA
ndave@csl.sri.com

Abstract. Development of computer aided verification tools has greatly benefited from SMT technologies; instead of writing an ad-hoc reasoning engine, designers translate their problem into SMT queries which solvers can efficiently solve. Translating a problem into effective SMT queries, however, is itself a tedious, error-prone, and non-trivial task. This paper introduces Smten, a tool for automatically translating high-level symbolic computations into SMT queries. We demonstrate the use of Smten in the development of an SMT-based string constraint solver.

1 Introduction

As Satisfiability Modulo Theories (SMT) solvers mature, their use continues to grow across many domains, including model checking, program synthesis, automated theorem proving, automatic test generation, and software verification. A primary reason for the popularity of SMT is it removes the need for ad-hoc reasoning engines in each application in favor of a simpler translation to a well understood domain with high-performance solvers.

Translating a problem into effective SMT queries, however, is itself a tedious, error-prone, and non-trivial task required for each new SMT-based tool. To better understand the effort involved in translating a problem into SMT queries, we will discuss issues that arise in the translation of Hampi [1], a string constraint solver originally implemented using the STP [2] SMT solver.

The primary form of string constraint supported by the Hampi solver is regular expression match. Figure 1 presents a high-level description in a Haskell-like pseudo-code of what it means to perform regular expression matches in Hampi. This description is polymorphic in the character type. At a high level, the goal is to compute the `match` function symbolically, given a regular expression and

* This work was sponsored by the Defense Advanced Research Projects Agency (DARPA) and the Air Force Research Laboratory (AFRL), under contract FA8750-10-C-0237 and supported by National Science Foundation under Grant No. CCF-1217498. The views, opinions, and/or findings contained in this report are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 678–683, 2013.

```
data RegEx = Epsilon | Empty | Atom Char | Range Char Char
           | Star RegEx | Concat RegEx RegEx | Or RegEx RegEx

match :: (SChar c) ⇒ RegEx → [c] → Bool
match Epsilon str       = null str
match Empty   _         = False
match (Atom x) [c]      = toSChar x == c
match (Range lo hi) [c] = toSChar lo ≤ c && c ≤ toSChar hi
match r@(Star x) str    = null str || any (match2 x r) (splits [1..length str] str)
match (Concat a b) str  = any (match2 a b) (splits [0..length str] str)
match (Or a b) str      = match a str || match b str

match2 a b (sa, sb) = match a sa && match b sb
splits ns x = map (λn → splitAt n x) ns
```

Fig. 1. High-level regular expression match

symbolic string, to obtain a boolean formula representing membership of the string in the language of the regular expression. This boolean formula is used in the SMT query.

The translation of regular expression match into a formula is complicated by the fact that the STP SMT solver does not support strings or characters. The symbolic string must first be translated into something understood by the SMT solver, *e.g.,* as a collection of free bit-vectors. High-level string operations must also be translated to SMT-understandable operations. Some operations, such as comparing two characters, translate naturally to bit-vector comparison in the SMT formula. Others, like string length, depend entirely on how strings are represented in the SMT query. Choosing how to represent high-level data types and operations in a lower-level SMT formula is often tedious.

In practice, directly translating high-level data types and operations into SMT formulas and querying the solver is not necessarily efficient. For example, if we know the length of a substring being matched against part of a regular expression, we can restrict the number of alternatives considered, drastically reducing the SMT query size and consequently the solver runtime. Similar improvements can be achieved by exploiting the knowledge of known character values in the string during the translation. Adding special code to exploit these cases is non-trivial, and it is not always obvious when it will lead to a worthwhile improvement in the translation process.

We have implemented Smten, a tool for automatically translating high-level symbolic computations into efficient SMT queries. Smten minimizes the manual effort of implementing and optimizing ad-hoc translations into SMT queries, leading to simpler, more readable code, and increasing developer productivity.

We demonstrate Smten via a new implementation of the Hampi string constraint solver. Smten allowed us to easily identify and implement optimizations in the SMT query generation, resulting in performance comparable to the original implementation in 5% of the code size.

```
data Symbolic a
instance Monad Symbolic

assert :: Bool → Symbolic ()

runSymbolic :: Symbolic a → IO (Maybe a)
```

```
class Free a where
    free :: Symbolic a

instance Free Bool
instance Free Integer
instance Free (Bit #n)
```

Fig. 2. The Smten `Symbolic` monad

Related Work

The value of augmenting SMT with general-purpose programming abstractions is well recognized [3–5] and can be found in multiple guises in the literature. SMT solvers, *e.g.,* Z3 [6] and Yices [7] add theories to improve the solver runtime, *e.g.,* record types and lambda terms; features which also raise the level of abstraction. For practical reasons, however, SMT developers have not devoted significant effort to abstractions aimed solely at improving the user's representation task, *e.g.,* modules, parametric and ad-hoc polymorphism, and metaprogramming.

In the context of using SMT solvers, multiple Domain Specific Embedded Languages (DSEL) [8] have been developed to hide the complexity of interacting with the SMT solver and provide a straightforward metaprogramming layer for SMT, *e.g.,* Haskell embeddings of Yices [9] and Z3 [10]. These allow a programmer to describe complicated SMT queries metaprogramatically using the host language. However, as these tend to focus on providing a syntactic bridge to the host language, they have issues naturally exposing SMT features which overlap with the host language abstractions, *e.g.,* user-defined data types.

Smten combines the metaprogramming of DSELs with the raised abstractions of SMT solvers to provide a more coherent user experience, allowing rich interactions between the two approaches.

2 Smten: Language and Implementation

The Smten input language is used to describe high-level symbolic computations for translation into SMT queries. Smten's input language is a strongly typed, purely functional language borrowing its syntax and many features from Haskell [11], including support for algebraic data types, pattern matching, polymorphism, and type classes. We chose to base the Smten language on Haskell because of its ability to concisely describe side-effect free computations. For a complete description of type classes, pattern matching, functions, and other Smten language features, we refer interested readers to the Haskell reference [11]. The remainder of this section is devoted to the `Symbolic` monad, the mechanism in Smten for managing symbolic computations.

Figure 2 summarizes the `Symbolic` monad in Smten. Computations in the `Symbolic` monad take place in the context of free variables and assertions. The primitives for using the `Symbolic` monad are described as follows:

`free` introduces a new free variable into the `Symbolic` context and returns an expression representing it. The expression returned can be used as a normal concrete expression in the Smten language. Smten provides primitive instances of `free` for types supported directly by the SMT solver and can automatically derive sensible, user-overloadable, instances of `free` for any bounded algebraic data types. Currently Smten provides primitive support for booleans, integers, and bit-vectors.

`assert` introduces a boolean assertion into the `Symbolic` context.

`runSymbolic` queries the solver to determine whether there exists an assignment to the free variables in the given symbolic object satisfying all assertions. If there is such an assignment, `runSymbolic` returns the value of its argument under that assignment, otherwise it returns `Nothing`.

Smten also provides primitives to enable incremental queries supported by many SMT solvers. These primitives are not discussed in this paper.

The high-level pseudocode for matching shown in Fig. 1 is valid Smten code. Given `match`, the `Symbolic` monad can be used to easily describe an SMT-based tool which accepts a length i and regular expression as input, and uses an SMT solver to find a concrete string of length i matching the regular expression:

```
main :: IO ()
main = do
  (len, regex) ← parseArgs
  result ← runSymbolic (qmatch len regex)
  case result of
    Just v → putStrLn v
    Nothing → putStrLn "no solution"
```

```
qmatch :: Integer → RegEx
                  → Symbolic String
qmatch len regex = do
  str ← sequence (replicate len free)
  assert (match regex str)
  return str
```

Note that the usage of the `match` function is the same whether the string argument is symbolic or concrete. Smten evaluates concrete inputs directly and generates SMT expressions for symbolic inputs.

The Smten tool compiles high-level descriptions of symbolic computations to Haskell using standard compilation techniques. Case expressions and primitive operations in the kernel language recognize concrete arguments and perform concrete evaluation wherever possible. This concrete evaluation removes objects with no primitive SMT support, such as lists and complex data structures, from the generated SMT query. Smten explicitly preserves dynamic sharing of expressions in the generated SMT queries.

As Smten generates Haskell, one can easily mix Haskell and Smten code, and, via Haskell's foreign function interface, other languages, *e.g.,* C or Java.

3 Implementing Hampi with Smten

Hampi is a string constraint solver whose constraints express membership of strings in both regular languages and fixed-size context-free languages. A Hampi input consists of regular expression and context free grammar (CFGs) definitions, bounded-size string variables, and predicates on these strings referencing

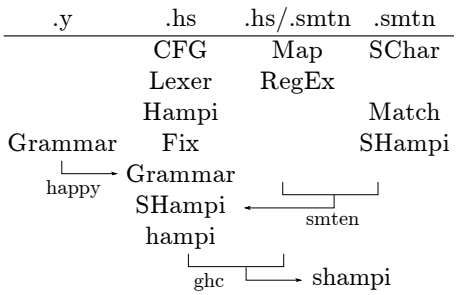


Fig. 3. SHAMPI source organization

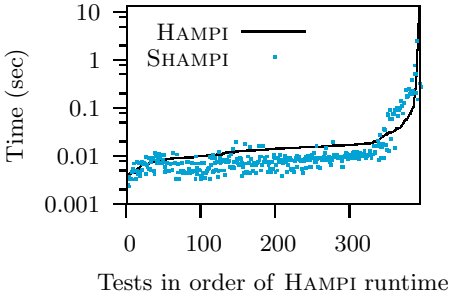


Fig. 4. HAMPI vs. SHAMPI runtime

the regular expressions and grammars. The output from HAMPI is a string which satisfies the constraints or a report that the constraints are unsatisfiable. The HAMPI tool has already been applied successfully to testing and analysis of real programs, most notably in static and dynamic analyses for SQL injections in web applications and automated bug finding in C programs using systematic testing. The original implementation of HAMPI was developed in about 20K lines of Java and uses the STP [2] SMT solver.

SHAMPI is our implementation of the HAMPI tool developed using Smten[1]. Figure 3 shows the organization of the source code for SHAMPI. We used the Happy parser generator to implement the HAMPI input parser. Much of the tool we left in Haskell, including the rest of the parser and fix-sizing of CFGs. The definition of RegEx is shared by both Haskell and Smten code. The match algorithm is implemented entirely in the Smten language. In total, our implementation of SHAMPI has a code base of 1030 lines.

Initially we used the naïve match algorithm from Fig. 1 for SHAMPI. To improve performance, we simplified CFGs by restricting them to match fixed-length strings. We further improved performance by caching boolean sub-match results in our match implementation. These optimizations, once understood, were implemented in Smten in a modest number of lines and in a matter of hours; replicating the same optimizations without Smten would have taken significantly more code and designer effort.

Figure 4 shows the performance of our implementation compared to the original implementation of HAMPI on all tests from the HAMPI distribution. For both SHAMPI and HAMPI, we took the best of 10 runs. SHAMPI was compiled with GHC-7.4.1 [12] and uses STP for solving SMT queries. We ran revision 46 of a single HAMPI server instance for all runs of all tests on HAMPI to amortize startup cost. Even so, SHAMPI outperforms HAMPI on most tests, and is within a factor of 8 in the worst case. Smten also allowed us to easily experiment with using other solvers and representations for symbolic characters. Our best

[1] Smten & SHAMPI source is at
`http://people.csail.mit.edu/ruhler/shampi.tar.gz`

variant represented characters as integers and called the Yices-2.1.0 [13] solver; it slightly improves runtime overall and reduces the worst case overhead to a factor of 4.

4 Conclusion

SMT technologies greatly benefit developers, allowing them to share a small set of high-performance solvers rather than develop their own ad-hoc reasoning engine. Smten further extends this sharing from actual computations to the translation of problems into SMT queries. Smten allows developers to leverage existing effort and expertise in the difficult task of translating problems to effective SMT queries.

Our Shampi example clearly illustrates the value of Smten; Shampi closely matches the performance of the Hampi tool, while being only 5% of the code.

We believe Smten has great potential in further improving SMT-based tool construction. In the future, we plan to explore naturally describing counter-example guided queries and extend the portfolio approach of SMT solving across multiple sets of theories and solvers.

References

1. Kiezun, A., Ganesh, V., Guo, P.J., Hooimeijer, P., Ernst, M.D.: Hampi: a solver for string constraints. In: Proceedings of the 18th International Symposium on Software Testing and Analysis, ISSTA 2009, pp. 105–116. ACM, New York (2009)
2. Ganesh, V., Dill, D.L.: A Decision Procedure for Bit-Vectors and Arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
3. Köksal, A.S., Kuncak, V., Suter, P.: Scala to the power of z3: Integrating smt and programming. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 400–406. Springer, Heidelberg (2011)
4. Agarwal, S.: Functional SMT solving: A new interface for programmers. Master's thesis, Indian Institute of Technology Kanpur (June 2012)
5. Barrett, C., Stump, A., Tinelli, C.: The Satisfiability Modulo Theories Library (SMT-LIB) (December 2010), `www.SMT-LIB.org`
6. de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Dutertre, B., Moura, L.D.: The yices smt solver (2006)
8. Leijen, D., Meijer, E.: Domain specific embedded compilers. In: Proceedings of the 2nd Conference on Domain-Specific Languages, pp. 109–122. ACM Press (1999)
9. Stewart, D.: `http://hackage.haskell.org/package/yices-painless-0.1.2` (2011)
10. Erkok, L.: `http://hackage.haskell.org/package/sbv-2.3` (July 2012)
11. Peyton Jones, S.: Haskell 98 Language and Libraries: the Revised Report (2003)
12. The Glasgow Haskell Compiler, `http://www.haskell.org/ghc`
13. (August 2012), `http://yices.csl.sri.com/index.shtml`

EXPLAIN: A Tool for Performing Abductive Inference

Isil Dillig and Thomas Dillig

Computer Science Department, College of William & Mary
{idillig,tdillig}@cs.wm.edu

Abstract. This paper describes a tool called EXPLAIN for performing abductive inference. Logical abduction is the problem of finding a simple explanatory hypothesis that explains observed facts. Specifically, given a set of premises Γ and a desired conclusion ϕ, abductive inference finds a simple explanation ψ such that $\Gamma \wedge \psi \models \phi$, and ψ is consistent with known premises Γ. Abduction has many useful applications in verification, including inference of missing preconditions, error diagnosis, and construction of compositional proofs. This paper gives a brief tutorial introduction to EXPLAIN and describes the basic inference algorithm.

1 Introduction

The fundamental ingredient of automated logical reasoning is *deduction*, which allows deriving valid conclusions from a given set of premises. For example, consider the following set of facts:

(1) $\forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x))$
(2) $\forall x.\ ((\text{duck}(x) \vee \text{goose(x)}) \Rightarrow \text{waddle}(x))$
(3) duck(donald)

Based on these premises, logical deduction allows us to reach the conclusion:

$$\text{waddle(donald)} \wedge \text{quack(donald)}$$

This form of forward deductive reasoning forms the basis of all SAT and SMT solvers as well as first-order theorem provers and verification tools used today.

A complementary form of logical reasoning to deduction is *abduction*, as introduced by Charles Sanders Peirce [1]. Specifically, abduction is a form of backward logical reasoning, which allows inferring likely premises from a given conclusion. Going back to our earlier example, suppose we know premises (1) and (2), and assume that we have observed that the formula $\text{waddle(donald)} \wedge \text{quack(donald)}$ is true. Here, since the given premises do not imply the desired conclusion, we would like to find an explanatory hypothesis ψ such that the following deduction is valid:

$$\frac{\begin{array}{c}\forall x.\ (\text{duck}(x) \Rightarrow \text{quack}(x)) \\ \forall x.\ ((\text{duck}(x) \vee \text{goose(x)}) \Rightarrow \text{waddle}(x)) \\ \psi\end{array}}{\text{waddle(donald)} \wedge \text{quack(donald)}}$$

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 684–689, 2013.

The problem of finding a logical formula ψ for which the above deduction is valid is known as *abductive inference*. For our example, many solutions are possible, including the following:

$$\begin{aligned}
&\psi_1 : \text{duck(donald)} \wedge \neg\text{quack(donald)}\\
&\psi_2 : \text{waddle(donald)} \wedge \text{quack(donald)}\\
&\psi_3 : \text{goose(donald)} \wedge \text{quack(donald)}\\
&\psi_4 \quad \text{duck(donald)}
\end{aligned}$$

While all of these solutions make the deduction valid, some of these solutions are more desirable than others. For example, ψ_1 contradicts known facts and is therefore a useless solution. On the other hand, ψ_2 simply restates the desired conclusion, and despite making the deduction valid, gets us no closer to explaining the observation. Finally, ψ_3 and ψ_4 neither contradict the premises nor restate the conclusion, but, intuitively, we prefer ψ_4 over ψ_3 because it makes fewer assumptions.

At a technical level, given premises Γ and desired conclusion ϕ, abduction is the problem of finding an explanatory hypothesis ψ such that:

$$\begin{aligned}
&(1)\ \Gamma \wedge \psi \models \phi\\
&(2)\ \Gamma \wedge \psi \not\models \text{false}
\end{aligned}$$

Here, the first condition states that ψ, together with known premises Γ, entails the desired conclusion ϕ. The second condition stipulates that ψ is consistent with known premises. As illustrated by the previous example, there are many solutions to a given abductive inference problem, but the most desirable solutions are usually those that are as simple and as general as possible.

Recently, abductive inference has found many useful applications in verification, including inference of missing function preconditions [2,3], diagnosis of error reports produced by verification tools [4], and for computing underapproximations [5]. Furthermore, abductive inference has also been used for inferring specifications of library functions [6] and for automatically synthesizing circular compositional proofs of program correctness [7].

In this paper, we describe our tool, called EXPLAIN, for performing logical abduction in the combination theory of Presburger arithmetic and propositional logic. The solutions computed by EXPLAIN are both simple and general: EXPLAIN always yields a logically weakest solution containing the fewest possible variables.

2 A Tutorial Introduction to EXPLAIN

The EXPLAIN tool is part of the SMT solver MISTRAL, which is available at http://www.cs.wm.edu/~tdillig/mistral under GPL license. MISTRAL is written in C++ and provides a C++ interface for EXPLAIN. In this section, we give a brief tutorial on how to solve abductive inference problems using EXPLAIN.

As an example, consider the abduction problem defined by the premises $x \leq 0$ and $y > 1$ and the desired conclusion $2x - y + 3z \leq 10$ in the theory of linear

```
 1. Term* x = VariableTerm::make("x");
 2. Term* y = VariableTerm::make("y");
 3. Term* z = VariableTerm::make("z");

 4. Constraint c1(x, ConstantTerm::make(0), ATOM_LEQ);
 5. Constraint c2(y, ConstantTerm::make(1), ATOM_GT);
 6. Constraint premises = c1 & c2;

 7. map<Term*, long int> elems;
 8. elems[x] = 2;
 9. elems[y] = -1;
10. elems[z] = 3;
11. Term* t = ArithmeticTerm::make(elems);
12. Constraint conclusion(t, ConstantTerm::make(10), ATOM_LEQ);

13. Constraint explanation = conclusion.abduce(premises);
14. cout << "Explanation: " << explanation << endl;
```

Fig. 1. C++ code showing how to use EXPLAIN for performing abduction

integer arithmetic. In other words, we want to find a simple formula ψ such that:

$$x \leq 0 \wedge y > 1 \wedge \psi \models 2x - y + 3z \leq 10$$
$$x \leq 0 \wedge y > 1 \wedge \psi \not\models \mathit{false}$$

Figure 1 shows C++ code for using EXPLAIN to solve the above abductive inference problem. Here, lines 1-12 construct the constraints used in the example, while line 13 invokes the `abduce` method of EXPLAIN for performing abduction. Lines 1-3 construct variables x, y, z, and lines 4 and 5 form the constraints $x \leq 0$ and $y > 1$ respectively. In MISTRAL, the operators `&, |, !` are overloaded and are used for conjoining, disjoining, and negating constraints respectively. Therefore, line 6 constructs the premise $x \leq 0 \wedge y > 1$ by conjoining `c1` and `c2`. Lines 7-12 construct the desired conclusion $2x - y + 3z \leq 10$. For this purpose, we first construct the arithmetic term $2x - y + 3z$ (lines 7-11). An `ArithmeticTerm` consists of a map from terms to coefficients; for instance, for the term $2x - y + 3z$, the coefficients of x, y, z are specified as $2, -1, 3$ in the `elems` map respectively.

The more interesting part of Figure 1 is line 13, where we invoke the `abduce` method to compute a solution to our abductive inference problem. For this example, the solution computed by EXPLAIN (and printed out at line 14) is $z \leq 4$. It is easy to confirm that $z \leq 4 \wedge x \leq 0 \wedge y > 1$ logically implies $2x - y + 3z \leq 10$ and that $z \leq 4$ is consistent with our premises.

In general, the abductive solutions computed by EXPLAIN have two theoretical guarantees: First, they contain as few variables as possible. For instance, in our example, although $z - x \leq 4$ is also a valid solution to the abduction problem, EXPLAIN always yields a solution with the fewest number of variables because such solutions are generally simpler and more concise. Second, among the class of solutions that contain the same set of variables, EXPLAIN always yields the *logically weakest* explanation. For instance, in our example, while $z = 0$ is also

a valid solution to the abduction problem, it is logically stronger than $z \leq 4$. Intuitively, logically weak solutions to the abduction problem are preferable because they make fewer assumptions and are therefore more likely to be true.

3 Algorithm for Performing Abductive Inference

In this section, we describe the algorithm used in EXPLAIN for performing abductive inference. First, let us observe that the entailment $\Gamma \wedge \psi \models \phi$ can be rewritten as $\psi \models \Gamma \Rightarrow \phi$. Furthermore, in addition to entailing $\Gamma \Rightarrow \phi$, we want ψ to obey the following three requirements:

1. The solution ψ should be consistent with Γ because an explanation that contradicts known premises is not useful
2. To ensure the simplicity of the explanation, ψ should contain as few variables as possible
3. To capture the generality of the abductive explanation, ψ should be no stronger than any other solution ψ' satisfying the first two requirements

Now, consider a *minimum satisfying assignment* (MSA) of $\Gamma \Rightarrow \phi$. An MSA of a formula φ is a partial satisfying assignment of φ that contains as few variables as possible. The formal definition of MSAs as well as an algorithm for computing them are given in [8]. Clearly, an MSA σ of $\Gamma \Rightarrow \phi$ entails $\Gamma \Rightarrow \phi$ and satisfies condition (2). Unfortunately, an MSA of $\Gamma \Rightarrow \phi$ does not satisfy condition (3), as it is a logically strongest solution containing a given set of variables.

Given an MSA of $\Gamma \Rightarrow \phi$ containing variables V, we observe that a logically weakest solution containing only V is equivalent to $\forall \overline{V}.\ (\Gamma \Rightarrow \phi)$, where $\overline{V} = \text{free}(\Gamma \Rightarrow \phi) - V$. Hence, given an MSA of $\Gamma \Rightarrow \phi$ consistent with Γ, an abductive solution satisfying all conditions (1)-(3) can be obtained by applying quantifier elimination to $\forall \overline{V}.\ (\Gamma \Rightarrow \phi)$.

Thus, to solve the abduction problem, what we want is a largest set of variables X such that $(\forall X.(\Gamma \Rightarrow \phi)) \wedge \Gamma$ is satisfiable. We call such a set of variables X a *maximum universal subset* (MUS) of $\Gamma \Rightarrow \phi$ with respect to Γ. Given an MUS X of $\Gamma \Rightarrow \phi$ with respect to Γ, the desired solution to the abductive inference problem is obtained by eliminating quantifiers from $\forall X.(\Gamma \Rightarrow \phi)$ and then simplifying the resulting formula with respect to Γ using the algorithm from [9].

Pseudo-code for our algorithm for solving an abductive inference problem defined by premises Γ and conclusion ϕ is shown in Figure 2. The `abduce` function given in lines 1-5 first computes an MUS of $\Gamma \Rightarrow \phi$ with respect to Γ using the helper `find_mus` function. Given such a maximum universal subset X, we obtain a quantifier-free abductive solution χ by applying quantifier elimination to the formula $\forall X.(\Gamma \Rightarrow \phi)$. Finally, at line 4, to ensure that the final abductive solution does not contain redundant subparts that are implied by the premises, we apply the simplification algorithm from [9] to χ. This yields our final abductive solution ψ which satisfies our criteria of minimality and generality and that is not redundant with respect to the original premises.

```
abduce(φ, Γ) {
1.  φ = (Γ ⇒ φ)
2.  Set X = find_mus(φ, Γ, free(φ), 0)
3.  χ = elim(∀X.φ)
4.  ψ = simplify(χ, Γ)
5.  return ψ
}

find_mus(φ, Γ, V, L) {
6.  If V = ∅ or |V| ≤ L  return ∅
7.  U = free(φ) − V
8.  if( UNSAT (Γ ∧ ∀U.φ))  return ∅

9.  Set best = ∅
10.  choose x ∈ V

11.  if(SAT(∀x.φ)) {
12.   Set Y = find_mus(∀x.φ, Γ, V \ {x}, L − 1);
13.   If (|Y| + 1 > L) { best = Y ∪ {x}; L = |Y| + 1 }
     }
14.  Set Y = find_mus(φ, Γ, V \ {x},L);
15. If (|Y| > L) { best = Y }

16. return best;
}
```

Fig. 2. Algorithm for performing abduction

The function `find_mus` used in `abduce` is shown in lines 6-16 of Figure 2. This algorithm directly extends the `find_mus` algorithm we presented earlier in [8] to exclude universal subsets that contradict Γ. At every recursive invocation, `find_mus` picks a variable x from the set of free variables in φ. It then recursively invokes `find_mus` to compute the sizes of the universal subsets with and without x and returns the larger universal subset. In this algorithm, L is a lower bound on the size of the MUS and is used to terminate search branches that cannot improve upon an existing solution. Therefore, the search for an MUS terminates if we either cannot improve upon an existing solution L, or the universal subset U at line 7 is no longer consistent with Γ. The return value of `find_mus` is therefore a largest set X of variables for which $\Gamma \wedge \forall X.\varphi$ is satisfiable.

4 Experimental Evaluation

To explore the size of abductive solutions and the cost of computing such solutions in practice, we collected 1455 abduction problems generated by the Compass program analysis system for inferring missing preconditions of functions. In each abduction problem $(\Gamma \wedge \psi) \Rightarrow \phi$, Γ represents known invariants, and ϕ is the weakest precondition of an assertion in some function f. Hence, the

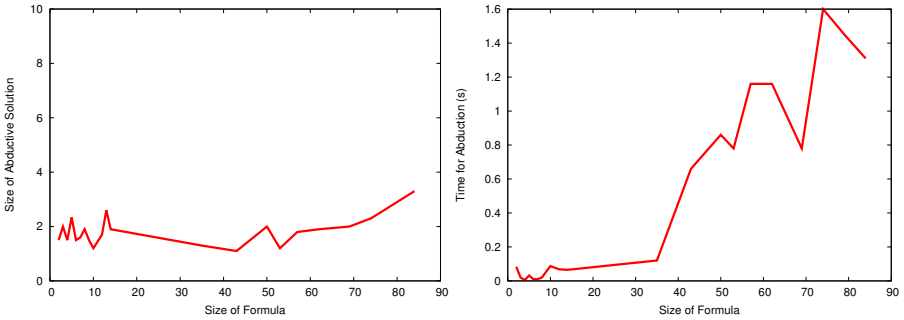


Fig. 3. Size of Formula vs. Size of Abductive Solution and Time for Abduction

solution ψ to the abduction problem represents a potential missing precondition of f sufficient to guarantee the safety of the assertion.

The left-hand side of Figure 3 plots the size of the formula $\Gamma \Rightarrow \phi$, measured as the number of leaves in the formula, versus the size of the computed abductive solution. As this graph shows, the abductive solution is generally much smaller than the original formula, demonstrating that our abduction algorithm generates small explanations in practice. The right-hand side of Figure 3 plots the size of the formula $\Gamma \Rightarrow \phi$ versus the time taken to solve the abduction problem. As expected, the time increases with formula size, but remains tractable even for the largest abduction problems in our benchmark set.

References

1. Peirce, C.: Collected papers of Charles Sanders Peirce. Belknap Press (1932)
2. Calcagno, C., Distefano, D., O'Hearn, P., Yang, H.: Compositional shape analysis by means of bi-abduction. POPL 44(1), 289–300 (2009)
3. Giacobazzi, R.: Abductive analysis of modular logic programs. In: Proceedings of the 1994 International Symposium on Logic Programming, Citeseer, pp. 377–391 (1994)
4. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)
5. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: POPL, pp. 235–246. ACM (2008)
6. Zhu, H., Dillig, I., Dillig, T.: Abduction-based inference of library specifications for source-sink property verification. In: Technical Report, College of William & Mary (2012)
7. Li, B., Dillig, I., Dillig, T., McMillan, K., Sagiv, M.: Synthesis of circular compositional program proofs via abduction. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 370–384. Springer, Heidelberg (2013)
8. Dillig, I., Dillig, T., McMillan, K.L., Aiken, A.: Minimum satisfying assignments for SMT. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 394–409. Springer, Heidelberg (2012)
9. Dillig, I., Dillig, T., Aiken, A.: Small formulas for large programs: On-line constraint simplification in scalable static analysis. In: Cousot, R., Martel, M. (eds.) SAS 2010. LNCS, vol. 6337, pp. 236–252. Springer, Heidelberg (2010)

A Tool for Estimating Information Leakage*

Tom Chothia, Yusuke Kawamoto, and Chris Novakovic

School of Computer Science, University of Birmingham, Birmingham, UK

Abstract. We present leakiEst, a tool that estimates how much information leaks from systems. To use leakiEst, an analyst must run a system with a range of secret values and record the outputs that may be exposed to an attacker. Our tool then estimates the amount of information leaked from the secret values to the observable outputs of the system. Importantly, our tool calculates the confidence intervals for these estimates, and tests whether they represent real evidence of an information leak in the system. leakiEst is freely available and has been used to verify the security of a range of real-world systems, including e-passports and Tor.

Introduction. Information leakage occurs when something about a system's secret data can be deduced from observing its public outputs. Not all information leakage is serious: many retailers' billing systems readily "leak" the last four digits of a credit card number, and password-checking functions "leak" some information about a secret password in response to an incorrect guess (e.g., that the guess is not the password). Information leakage is therefore quantitative and it is important to be able to answer the question "how much information does a system leak?". Information theory is a useful framework for quantifying these leaks in systems (see e.g. [9]), and two particular measures, mutual information and min-entropy leakage, place useful bounds on an attacker's ability to guess the secrets from the public outputs.

Our tool, leakiEst, estimates these leakage measures from datasets containing secrets and public outputs that are generated from trial runs of a system. Its methodology is based on our previous work that provides rigorous verification methods for estimating information leakage [3,4]; it performs statistical tests to distinguish an insecure system with a very small information leak from a secure one with no leaks. This is similar to detecting a correlation between two random variables, a well-investigated problem, and we compare leakiEst's performance to that of existing statistical tests. If a leak is found, leakiEst can display the conditional probability of observing each output from the system given a particular secret, which may be used to derive a concrete attack against the system.

There are several tools that calculate the amount of information that leaks from a program (e.g., [7,2]). These tools provide tight bounds, but require access to the source code of the program and a formalism that is powerful enough to model the underlying system. These requirements are often prohibitive, and

* This work was supported by EPSRC Research Grant EP/J009075/1.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 690–695, 2013.

prevent these tools from being used in the case studies below. By estimating leakage based on trial runs of a system, we trade precise leakage calculations for the ability to detect leaks in complex, real-world systems. Other tools, such as Weka [6], can estimate the mutual information of random variables, but do not calculate the confidence interval for their estimates, nor do they test for compatibility with zero leakage; it is therefore difficult to ensure that the estimates produced by these other tools are meaningful.

leakiEst, its documentation and sample datasets are available at [1].

Estimating Information Leakage. leakiEst can analyse data collected from systems that contain a secret value, and whose observable behaviour is probabilistic and possibly affected by the secret. We assume that there is a probability distribution X on the secret values and another probability distribution Y on the public outputs. The system is defined by the likelihood of observing each possible output given each possible secret, i.e., by the conditional probability distribution $P_{Y|X}$. As an example we consider a Java program in which two random integers between 1 and 10 are generated sequentially; the first is visible to an attacker, and the second must be kept secret. One potential flaw would be the use of the Java API's cryptographically weak `Random` class for pseudorandom number generation, rather than the stronger `SecureRandom` class: if `Random` were used, the value of the second integer may be related to the first, which would constitute an information leak. In this case, there would be a correlation between Y, the probability distribution on the integer that the attacker observes, and X, the distribution on the secret integer generated afterwards. X and Y's mutual information, or the min-entropy leakage from X to Y, defines how difficult it is for the attacker to guess the secret integer from the observable integer.

leakiEst estimates the magnitude of information leaks solely from trial runs of a system: a user must first run a system many times with a range of possible secret values and record the observed outputs to create a dataset that can be processed by leakiEst. This system-agnostic approach offers the greatest flexibility to users of the tool, and we note that for particular types of systems (e.g., RFID cards, web traffic, or Java programs) it would be possible to build a framework to automatically generate an appropriate dataset; we provide a Java API so leakiEst's functionality can easily be integrated with other tools.

Given a dataset, the tool estimates the conditional probability distribution $\hat{P}_{Y|X}$ of the system and implements tests we have proposed in previous work [3,4]. In [3] we calculate the bias and distribution that the repeated estimates of discrete mutual information will follow in terms of the true mutual information. This allows us to estimate a confidence interval, and therefore test whether the apparent leakage indicates a statistically significant information leak in the system or whether it is in fact consistent with zero leakage. The estimates are non-parametric; i.e., they do not assume that secrets and outputs fit any particular distribution. In [4] we extend this technique to cover estimates of continuous mutual information using kernel density estimation, allowing leakiEst to estimate leakage from systems with continuous outputs.

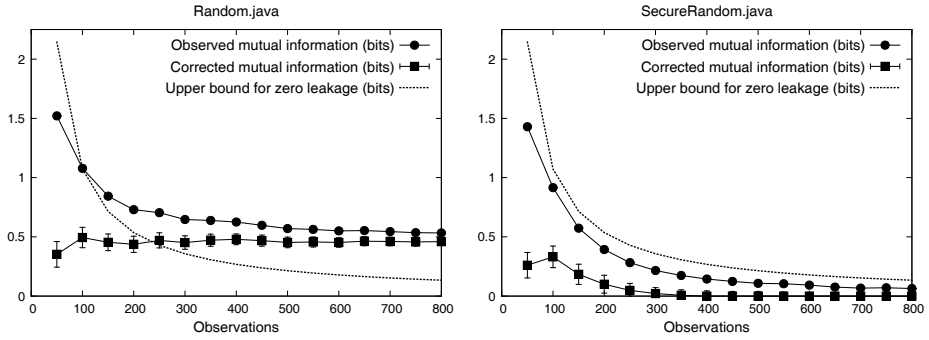


Fig. 1. A depiction of leakiEst's output for the Java `Random` and `SecureRandom` sample programs. The graphs' data sources are produced by leakiEst's `-csv` option, which calculates leakage estimates at user-defined intervals in a dataset and writes them to a CSV file.

Two further novel features of our tool are the calculation of an upper bound on the expected mutual information for systems containing no information leaks and the calculation of confidence intervals. It is these features that allow us to interpret leakiEst's output in a meaningful way. Fig. 1 depicts leakiEst's output for datasets generated by programs that utilise the Java `Random` and `SecureRandom` classes in the manner described earlier (their source code is available at [1]). Each graph denotes both the observed and corrected mutual information of the secret values and public outputs in the dataset, with the x-axis denoting the number of observations that produce those estimates. Other tools (e.g., Weka) can calculate the uncorrected observed mutual information value, but this is of limited use in isolation, as it can be difficult to tell whether that value represents a true information leak or is just noise in the data.

The dashed line shows the 95% upper bound on the measurement expected if the true mutual information were zero. For values of $x > 150$, when the observed mutual information falls below the upper bound for zero leakage, the left-hand graph provides clear evidence that there is an information leak from the first random integer to the second when the `Random` class is used to generate random numbers. For cases where the mutual information is not zero (i.e., there is a leak), [3] provides a prediction of the bias and variance of the estimate. This allows us to make a prediction of the true mutual information, labelled as "corrected mutual information" in both graphs. This quantifies the information leakage (approximately 0.5 bits) accurately, even for a very small number of observations. The right-hand graph shows that when `SecureRandom` is used to generate random numbers the mutual information is always below the expected confidence interval for zero leakage, therefore there is no evidence of leakage from the first random integer to the second. While this does not guarantee that `SecureRandom` is secure (and for much larger ranges of numbers it may not be), the data processing inequality guarantees that an attacker learns nothing statistically significant from this particular dataset.

Our estimates make the assumption that terms of the order $(\#\text{samples})^{-2}$ are small, but for a large number of secrets and outputs and a small number of samples this may not be the case. For instance, when the product of the number of secrets and outputs is in the hundreds, tens of thousands of samples may be required for reliable results. leakiEst can test whether enough samples have been provided and warn the user if more are required.

leakiEst uses a new technique [1] to calculate the confidence interval for the estimated min-entropy leakage, which measures the vulnerability of secrets to single-attempt guessing attacks.

The Tool. leakiEst is developed in Java and may be used as either a Java API or as a standalone JAR file that can be invoked from the command line, so it can be integrated into the development workflow of software written in any programming language; it is a suitable component of system testing (to uncover new information leaks) as well as regression testing (to ensure that previously-discovered leaks have not been reintroduced). leakiEst's Java API also exposes common information theory and statistical functions that developers may find useful. We chose Java, as opposed to (e.g.) MATLAB or R, to make the tool more accessible to non-specialists and to simplify standalone execution.

The simplest datasets processed by leakiEst are text files with lines of the form `("secret","output")` describing a single trial run of the system. From this input, the tool calculates the conditional probability distribution $\hat{P}_{Y|X}$, estimates min-entropy leakage and mutual information using $\hat{P}_{Y|X}$, calculates the confidence intervals, and (for discrete mutual information) performs tests on the estimate in search of statistical evidence of non-zero leakage. For more complex systems (e.g., those containing multi-part secrets or producing multiple outputs per secret), leakiEst can process datasets recorded in Weka's ARFF file format. While each line of the file still describes a single trial run, the format of each line is more structured, with named "attributes" allowing the system's various secrets and outputs to be distinguished. Command-line options can be supplied to instruct leakiEst to treat arbitrary sets of attributes as secrets or outputs. Given an ARFF file, the tool calculates the mutual information and min-entropy leakage confidence intervals for the specified secret and each of the outputs individually. Another function orders all of the outputs by the amount of information they leak, allowing users to focus their attention on minimising leakage caused by particular outputs.

Scalability. leakiEst generates the conditional probability matrix for $\hat{P}_{Y|X}$ for a given list of observations. The tool updates the matrix in-place for each observation read, meaning it scales well for datasets containing a large number of observations: our `Random` and `SecureRandom` datasets, each containing 500 million observations forming 10×10 matrices, can be analysed in 3 minutes on a modern desktop computer. The tool scales less well for datasets containing a large number of unique secrets and outputs (which result in matrices with larger dimensions), but is nevertheless able to estimate leakage in many real-world scenarios: systems where the secret is a 4-digit PIN and the output is a binary value

Table 1. The p-values of leakiEst and other non-parametric tests when applied to an e-passport dataset containing 500 observations

Nationality	leakiEst	KS test	CVM test	AD test	BWS test
British	0	0	0	0	0
Irish	0	0.001	0	0	0
Greek	0.075	0.718	0.544	0.367	0.408
German	0	0.257	0.743	0.302	0.271

(i.e., a $\approx 2^{13} \times 2$ matrix) can be analysed in under 10 seconds, and systems where the secret is 19 bits of a key and the output is a binary value (i.e., a $2^{19} \times 2$ matrix) can be analysed in a day.

Case Study: Fixing an e-Passport Traceability Attack. The RFID chip in e-passports is designed to be untraceable; i.e., without knowing the secret key for a passport, it should be impossible to distinguish it from another passport across sessions. In [4,5] we observed that e-passports fail to achieve this goal due to a poorly-implemented MAC check: passports take longer to reject replayed messages. This means that a single message can be used to test for the presence of a particular passport. Here, the secret is a binary value indicating whether the passport is the one the attacker is attempting to trace, and the output is the time taken for the passport to reply. We collected timing data from an e-passport and analysed it with leakiEst, which clearly detects the presence of an information leak from a dataset containing 100 observations. Attempting to fix the leak, we developed a variant of the e-passport protocol that pads the time delays so that the average response time is equal in all cases [4]. leakiEst still indicated the presence of a small information leak: while the average times are the same, it appears that the actual time measurements come from a different distribution. After modifying the protocol to continue processing a message even when the MAC check fails, and only reject it at the end of the protocol, leakiEst indicates that it is free from leaks.

In cases where the secret is a single binary value, a number of existing non-parametric tests can be used to test whether two samples originate from the same distribution. The most popular of these are the Kolmogorov-Smirnov (KS), Baumgartner-Weiß-Schindler (BWS), Anderson-Darling (AD) and Cramér-von Mises (CVM) tests. Table 1 compares the p-values of these tests when applied to 500 observations of the time-padded protocol variant for e-passports from a range of countries that all implement different variations of the protocol. The p-value indicates the proportion of tests that failed to detect the leak, so the table shows that leakiEst detects leaks more reliably than the other tests for datasets of this size.

Case Study: Fingerprinting Tor Traffic. Tor is an anonymity system that uses encryption and onion routing to disguise users' network traffic. Traffic is encrypted before it leaves the user's node, but an intermediary can still infer

information about the web sites a user is requesting based on characteristics of the encrypted traffic: the time taken to respond to the request, the number of packets, the packet sizes, the number of "spikes" in the data stream, etc. Here, the secret is the URL of the web site whose encrypted traffic is being intercepted by the attacker, and the public outputs are the characteristics of the encrypted traffic that the attacker is able to observe.

Such an attack has previously been mounted against Tor [8], using Weka to fingerprint encrypted traffic with a 54% success rate; since fingerprinting the web site is possible, clearly a leak exists. We generated a dataset by accessing each of the Alexa top 500 web sites ten times through a Tor node and recording features of the encrypted traffic. leakiEst ranks them and identifies which features (or sets of features) leak information. Existing machine learning tools (e.g., Weka) can be configured to select features based on mutual information but, uniquely, leakiEst estimates a confidence interval for each measure of mutual information, ranks the features in the dataset by reliability, and identifies the features that do not leak information. leakiEst's analysis showed that a web site is most easily fingerprinted by the number of spikes in the data stream. It also showed that some features suggested by previous authors, such as the average packet size, do not contain any useful information; we verified this by removing these features from the dataset and rerunning the classification in Weka, and observed no drop in the identification rate.

References

1. leakiEst, `http://www.cs.bham.ac.uk/research/projects/infotools/leakiest/`
2. Backes, M., Köpf, B., Rybalchenko, A.: Automatic Discovery and Quantification of Information Leaks. In: Proc. S&P, pp. 141–153 (2009)
3. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical Measurement of Information Leakage. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 390–404. Springer, Heidelberg (2010)
4. Chothia, T., Guha, A.: A Statistical Test for Information Leaks Using Continuous Mutual Information. In: Proc. CSF, pp. 177–190 (2011)
5. Chothia, T., Smirnov, V.: A Traceability Attack against e-Passports. In: Sion, R. (ed.) FC 2010. LNCS, vol. 6052, pp. 20–34. Springer, Heidelberg (2010)
6. Hall, M., Frank, E., Holmes, G., Pfahringer, B., Reutemann, P., Witten, I.H.: The WEKA Data Mining Software. SIGKDD Explorations 11(1), 10–18 (2009)
7. McCamant, S., Ernst, M.D.: Quantitative Information Flow as Network Flow Capacity. In: Proc. PLDI, pp. 193–205 (2008)
8. Panchenko, A., Niessen, L., Zinnen, A., Engel, T.: Website Fingerprinting in Onion Routing Based Anonymization Networks. In: Proc. WPES, pp. 103–114 (2011)
9. Smith, G.: On the Foundations of Quantitative Information Flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)

The TAMARIN Prover for the Symbolic Analysis of Security Protocols

Simon Meier[1], Benedikt Schmidt[2], Cas Cremers[1], and David Basin[1]

[1] Institute of Information Security, ETH Zurich, Switzerland
[2] IMDEA Software Institute, Madrid, Spain

Abstract. The TAMARIN prover supports the automated, unbounded, symbolic analysis of security protocols. It features expressive languages for specifying protocols, adversary models, and properties, and support for efficient deduction and equational reasoning. We provide an overview of the tool and its applications.

1 Introduction

During the last two decades, there has been considerable research devoted to the symbolic analysis of security protocols and existing tools have had considerable success both in detecting attacks on protocols and showing their absence. Nevertheless, there is still a large discrepancy between the symbolic models that one specifies on paper and the models that can be effectively analyzed by tools.

In this paper, we present the TAMARIN prover for the symbolic analysis of security protocols. TAMARIN generalizes the backwards search used by the Scyther tool [1] to enable: protocol specification by multiset rewriting rules; property specification in a guarded fragment of first-order logic, which allows quantification over messages and timepoints; and reasoning modulo equational theories. As practical examples, these generalizations respectively enable the tool to handle: protocols with non-monotonic mutable global state and complex control flow such as loops; complex security properties such as the eCK model [2] for key exchange protocols; and equational theories such as Diffie-Hellman, bilinear pairings, and user-specified subterm-convergent theories.

TAMARIN provides two ways of constructing proofs: an efficient, fully automated mode that uses heuristics to guide proof search, and an interactive mode. If the tool's automated proof search terminates, it returns either a proof of correctness (for an unbounded number of threads and fresh values) or a counterexample (e. g., an attack). Due to the undecidable nature of most properties in our setting, the tool may not terminate. The interactive mode enables the user to explore the proof states, inspect attack graphs, and seamlessly combine manual proof guidance with automated proof search.

The theory for Diffie-Hellman exponentiation and the application to Diffie-Hellman-based two-party key exchange protocols have been published in [3]. In the theses of Meier [4] and Schmidt [5], the approach is extended with trace induction and with support for bilinear pairings and AC operators.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 696–701, 2013.

2 Tool Description

We first give an example that illustrates TAMARIN's use. Afterwards, we describe its underlying foundations and implementation.

2.1 Example: Diffie-Hellman

Input. TAMARIN takes as its command-line input the name of a theory file that defines the equational theory modeling the protocol messages, the multiset rewriting system modeling the protocol, and a set of lemmas specifying the protocol's desired properties. To analyze the security of a variant of the Diffie-Hellman protocol, we use a theory file that consists of the following parts.

Equational Theory. To specify the set of protocol messages, we use:

```
builtins: diffie-hellman
functions: mac/2, g/0, shk/0 [private]
```

This enables support for Diffie-Hellman (DH) exponentiation and defines three function symbols. The support for DH exponentiation defines the operator `^` for exponentiation, which satisfies the equation $(g \hat{} x) \hat{} y = (g \hat{} y) \hat{} x$, and additional operators and equations. We use the binary function symbol `mac` to model a message authentication code (MAC), the constant `g` to model the generator of a DH group, and the constant `shk` to model a shared secret key, which is declared as private and therefore not directly deducible by the adversary. Support for pairing and projection using `<_,_>`, `fst`, and `snd` is provided by default.

Protocol. Our protocol definition consists of three (labeled) multiset rewriting rules. These rules have sequences of facts as left-hand-sides, labels, and right-hand-sides, where facts are of the form $F(t_1, \ldots, t_k)$ for a fact symbol F and terms t_i. The protocol rules use the fixed unary fact symbols `Fr` and `In` in their left-hand-side to obtain fresh names (unique and unguessable constants) and messages received from the network. To send a message to the network, they use the fixed unary fact symbol `Out` in their right-hand-side.

Our first rule models the creation of a new protocol thread *tid* that chooses a fresh exponent x and sends out g^x concatenated with a MAC of this value and the participants' identities:

```
rule Step1: [ Fr(tid:fresh), Fr(x:fresh) ] –[  ]→
            [ Out(<g^(x:fresh), mac(shk, <g^(x:fresh), A:pub, B:pub>)>)
            , Step1(tid:fresh, A:pub, B:pub, x:fresh) ]
```

In this rule, we use the sort annotations `:fresh` and `:pub` to ensure that the corresponding variables can only be instantiated with fresh and public names. An instance of the `Step1` rule rewrites the state by *consuming* two `Fr`-facts to obtain the fresh names `tid` and `x` and *generating* an `Out`-fact with the sent message and a `Step1`-fact denoting that given thread has completed the first step with the given parameters. The arguments of `Step1` denote the thread identifier, the actor, the intended partner, and the chosen exponent. The rule is always silent since there is no label.

Our second rule models the second step of a protocol thread:

```
rule Step2: [ Step1(tid, A, B, x:fresh), In(<Y, mac(shk, <Y, B, A>)>) ]
            –[ Accept(tid, Y^(x:fresh)) ]→ []
```

Here, a `Step1`-fact, which must have been created in an earlier `Step1`-step, is consumed in addition to an `In`-fact. The `In`-fact uses pattern matching to verify the MAC. The corresponding label `Accept(tid, Y^(x:fresh))` denotes that the thread `tid` has accepted the session key `Y^(x:fresh)`.

Our third rule models revealing the shared secret key to the adversary:

```
rule RevealKey: [] –[ Reveal() ]→ [ Out(shk) ]
```

The constant `shk` is output on the network and the label `Reveal()` ensures that the trace reflects *if* and *when* a reveal happens.

The set of protocol traces is defined via multiset rewriting (modulo the equational theory) with these rules and the builtin rules for fresh name creation, message reception by the adversary, message deduction, and message sending by the adversary, which is observable via facts of the form `K(m)`. More precisely, the trace corresponding to a multiset rewriting derivation is the sequence of the labels of the applied rules.

Properties. We define the desired security properties of the protocol as trace properties. The labels of the protocol rules must therefore contain enough information to state these properties. In Tamarin, properties are specified as lemmas, which are then discharged or disproven by the tool.

```
lemma Accept_Secret:
  ∀ i j tid key. Accept(tid,key)@i & K(key)@j ⇒ ∃ l. Reveal()@l & l < i
```

The lemma quantifies over timepoints i, j, and l and messages *tid* and *key*. It uses predicates of the form $F@i$ to denote that the trace contains the fact F at index i and predicates of the form $i<j$ to denote that the timepoint i is smaller than the timepoint j. The lemma states that if a thread `tid` has accepted a key `key` at timepoint `i` and `key` is also known to the adversary, then there must be a timepoint `l` before `i` where the shared secret was revealed.

Output. Running Tamarin on this input file yields the following output.

```
analyzed example.spthy: Accept_Secret (all-traces) verified (9 steps)
```

The output states that Tamarin successfully verified that all protocol traces satisfy the formula in `Accept_Secret`.

2.2 Theoretical Foundations

A formal treatment of Tamarin's foundations is given in the theses of Schmidt [5] and Meier [4]. For an equational theory E, a multiset rewriting system R defining a protocol, and a guarded formula φ defining a trace property, Tamarin can either check the validity or the satisfiability of φ for the traces of R modulo E. As usual, validity checking is reduced to checking the satisfiability of the negated formula. Here, constraint solving is used to perform an exhaustive, symbolic search for executions with satisfying traces. The states of the search

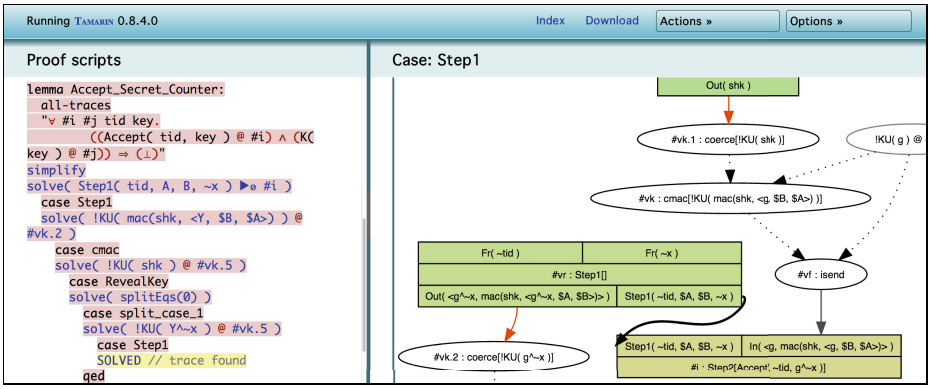


Fig. 1. TAMARIN's interactive mode

are constraint systems. For example, a constraint can express that some multiset rewriting step occurs in an execution or that one step occurs before another step. We can also directly use formulas as constraints to express that some behavior does *not occur* in an execution. Applications of constraint reduction rules, such as simplifications or case distinctions, correspond to the incremental construction of a satisfying trace. If no further rules can be applied and no satisfying trace was found, then no satisfying trace exists. For symbolic reasoning, we exploit the finite variant property [6] to reduce reasoning modulo E with respect to R to reasoning modulo AC with respect to the variants of R.

2.3 Implementation and Interactive Mode

TAMARIN is written in the Haskell programming language. Its interactive mode is implemented as a webserver, serving HTML pages with embedded Javascript. The source code of TAMARIN is publicly available from its webpage [7]. Figure 1 shows TAMARIN's interactive mode, which integrates automated analysis and interactive proof guidance, and provides detailed information about the current constraints or counterexample traces. Users can carry out automated analysis of parts of the search space and perform partial unfoldings of the proof tree.

3 Experimental Results

TAMARIN's flexible modeling framework and expressive property language make it suitable for analyzing a wide range of security problems. Table 1 shows selected results when using TAMARIN in the automated mode. These results illustrate TAMARIN's scope and effectiveness at unbounded verification and falsification.

Key Exchange Protocols. We used TAMARIN to analyze many authenticated key exchange protocols with respect to their intended adversary models [3]. These protocols typically include Diffie-Hellman exponentiation and are designed to satisfy complex security properties, such as the eCK model [2]. Earlier works had

Table 1. Selected results of the automated analysis of case studies included in the public TAMARIN repository. Here, KI denotes key independence.

	Protocol	Security property	Result	Time [s]	Details in
1.	KAS1	KI with Key Compromise Impersonation	proof	0.7	[3]
2.	NAXOS	eCK	proof	4.4	[3]
3.	STS-MAC	KI, adversary can register arbitrary public keys	attack	4.6	[3]
4.	STS-MAC-fix1	KI, adversary can register arbitrary public keys	proof	9.2	[3]
5.	STS-MAC-fix2	KI, adversary can register arbitrary public keys	proof	1.8	[3]
6.	TS1-2004	KI	attack	0.3	[3]
7.	TS2-2004	KI with weak Perfect Forward Secrecy	attack	0.5	[3]
8.	TS3-2004	KI with weak Perfect Forward Secrecy	non-termination	-	[3]
9.	UM	Perfect Forward Secrecy	attack	1.5	[3]
10.	TLS handshake	secrecy, injective agreement	proof	2.3	[4]
11.	TESLA 1	data authenticity	proof	4.4	[4]
12.	TESLA 2 (lossless)	data authenticity	proof	16.4	[4]
13.	Keyserver	keys are secret or revoked	proof	0.1	[4]
14.	Security Device	exclusivity (left or right)	proof	0.4	[4]
15.	Contract signing protocol	exclusivity (abort or resolve)	proof	0.8	[4]
16.	Envelope (no reboot)	denied access implies secrecy	proof	32.7	[4]
17.	SIGJOUX (tripartite)	Perfect Forward Secrecy	proof	102.9	[5]
18.	SIGJOUX (tripartite)	Perfect Forward Secrecy, ephemeral-key reveal	attack	111.5	[5]
19.	RYY (ID-based)	Perfect Forward Secrecy	proof	10.3	[5]
20.	RYY (ID-based)	Perfect Forward Secrecy, ephemeral-key reveal	attack	10.5	[5]
21.	YubiKey (multiset)	injective authentication	proof	19.3	[11]
22.	YubiHSM (multiset)	injective authentication	proof	7.6	[11]

only considered some of these protocols with respect to weaker adversaries, which cannot reveal random numbers and both short-term and long-term keys. The SIGJOUX and RYY protocols use bilinear pairings, which require a specialized equational theory that extends the Diffie-Hellman theory.

Loops and Mutable Global State. We also used TAMARIN to analyze protocols with loops and non-monotonic mutable global state. Examples include the TESLA protocols, the security device and contract signing examples from [8], the keyserver protocol from [9], and the exclusive secrets and envelope protocol models for TPMs from [10]. In each case, our results are more general or the analysis is more efficient than previous results. Additionally, TAMARIN was successfully used to analyze the YubiKey and YubiHSM protocols [11].

4 Related Tools

There are many tools for the symbolic analysis of security protocols. We focus on those that can provide verification with respect to an unbounded number of sessions for complex properties. In general, the TAMARIN prover offers a novel combination of features that enables it to verify protocols and properties that were previously impossible using other automated tools.

Like its predecessor the Scyther tool [1], TAMARIN performs backwards reasoning. However in contrast to Scyther, it supports equational theories, modeling complex control flow and mutable global state, an expressive property specification language, and the ability to combine interactive and automated reasoning.

The Maude-NPA tool [12] supports protocols specified as linear role-scripts, properties specified as symbolic states, and equational theories with a finite

variant decomposition modulo AC, ACI, or C. It is unclear if our case studies that use global state, loops, and temporal formulas can be specified in Maude-NPA. With respect to their support of equational theories, Maude-NPA and TAMARIN are incomparable. For example, Maude-NPA has been applied to XOR and TAMARIN has been applied to bilinear pairing.

The ProVerif tool [13] has been extended to partially handle DH with inverses [14], bilinear pairings [15], and mutable global state [8]. From a user perspective, TAMARIN provides a more expressive property specification language that, e. g., allows for direct specification of temporal properties. The effectiveness of ProVerif relies largely on its focus on the adversary's knowledge. It has more difficulty dealing with properties that depend on the precise state of agent sessions and mutable global state. The extension [8] for mutable global state is subject to several restrictions and the protocol models require additional manual abstraction steps. Similarly, the DH and bilinear pairing extensions work under some restrictions, e. g., exponents in the specification must be ground.

References

1. Cremers, C.J.F.: The Scyther Tool: Verification, falsification, and analysis of security protocols. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 414–418. Springer, Heidelberg (2008)
2. LaMacchia, B., Lauter, K., Mityagin, A.: Stronger security of authenticated key exchange. In: Susilo, W., Liu, J.K., Mu, Y. (eds.) ProvSec 2007. LNCS, vol. 4784, pp. 1–16. Springer, Heidelberg (2007)
3. Schmidt, B., Meier, S., Cremers, C., Basin, D.: Automated analysis of Diffie-Hellman protocols and advanced security properties. In: Proc. CSF. IEEE (2012)
4. Meier, S.: Advancing Automated Security Protocol Verification. PhD thesis (2013)
5. Schmidt, B.: Formal Analysis of Key Exchange Protocols and Physical Protocols. PhD thesis (2012)
6. Comon-Lundh, H., Delaune, S.: The finite variant property: How to get rid of some algebraic properties. In: Giesl, J. (ed.) RTA 2005. LNCS, vol. 3467, pp. 294–307. Springer, Heidelberg (2005)
7. `http://www.infsec.ethz.ch/research/software/tamarin`
8. Arapinis, M., Ritter, E., Ryan, M.: Statverif: Verification of stateful processes. In: Proc. CSF. IEEE (2011)
9. Mödersheim, S.: Abstraction by set-membership: verifying security protocols and web services with databases. In: Proc. CCS, pp. 351–360. ACM (2010)
10. Delaune, S., Kremer, S., Ryan, M.D., Steel, G.: Formal analysis of protocols based on TPM state registers. In: Proc. CSF, pp. 66–80. IEEE (2011)
11. Künnemann, R., Steel, G.: YubiSecure? Formal security analysis results for the YubiKey and YubiHSM. In: Jøsang, A., Samarati, P., Petrocchi, M. (eds.) STM 2012. LNCS, vol. 7783, pp. 257–272. Springer, Heidelberg (2013)
12. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer and its meta-logical properties. TCS 367, 162–202 (2006)
13. Blanchet, B.: An efficient cryptographic protocol verifier based on Prolog rules. In: Proc. CSFW. IEEE (2001)
14. Küsters, R., Truderung, T.: Reducing protocol analysis with xor to the xor-free case in the Horn theory based approach. J. Autom. Reasoning 46(3-4), 325–352 (2011)
15. Pankova, A., Laud, P.: Symbolic analysis of cryptographic protocols containing bilinear pairings. In: Proc. CSF. IEEE (2012)

QUAIL: A Quantitative Security Analyzer for Imperative Code*

Fabrizio Biondi[1], Axel Legay[2], Louis-Marie Traonouez[2], and Andrzej Wąsowski[1]

[1] IT University of Copenhagen, Denmark
[2] INRIA Rennes, France

Abstract. Quantitative security analysis evaluates and compares how effectively a system protects its secret data. We introduce QUAIL, the first tool able to perform an arbitrary-precision quantitative analysis of the security of a system depending on private information. QUAIL builds a Markov Chain model of the system's behavior as observed by an attacker, and computes the correlation between the system's observable output and the behavior depending on the private information, obtaining the expected amount of bits of the secret that the attacker will infer by observing the system. QUAIL is able to evaluate the safety of randomized protocols depending on secret data, allowing to verify a security protocol's effectiveness. We experiment with a few examples and show that QUAIL's security analysis is more accurate and revealing than results of other tools.

1 Introduction

The Challenge. Qualitative analysis tools can verify the complete security of a protocol, i.e. that an attacker is unable to get any information on a secret by observing the system—a property known as *non-interference*. Non-interference holds when the system's output is independent from the value of the secret, so no information about the latter can be inferred from the former [20]. However, when non-interference does not hold, qualitative analysis cannot rank the security of a system: all unsafe systems are the same.

Quantitative analysis can be used to decide which of two alternative protocols is more secure. It can also asses security of systems that are insecure, but nevertheless useful, in the qualitative sense, such as a password authentication protocol, for which there is always a positive probability that an attacker will randomly guess the password. A quantitative analysis is challenging because it is not sufficient to find a counterexample to a specification to terminate. We need to analyze all possible behaviors of the system and quantify for each one the probability that it will happen and how much of the protocol's secret will be revealed. So far no tool was able to perform this analysis precisely.

Quantitative analysis with QUAIL. We use Quantified Information Flow to reduce the comparison of security of two systems to a computation of expected amount of information, in the information-theoretical sense, that an attacker would learn about the secret by observing a system's behavior. This expected amount of information is known as

* Partially supported by MT-LAB — VKR Centre of Excellence on Modeling of IT.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 702–707, 2013.

information leakage [9,15,8,12,21] of a system. It amounts to zero iff the system is non-interfering [15], else it represents the expected number of bits of the secret that the attacker is able to infer. The analysis generalizes naturally to more than two systems, hence allowing to decide which of them is less of a threat to the secrecy of the data.

To compute information leakage we use a stochastic model of a system as observed by the attacker. The model is obtained by resolving non-determinism in the system code, using the prior probability distribution over the secret values known to the attacker before an attack. Existing techniques represent this with a channel matrix from secret values to outputs [5]. They build a row of the channel matrix for each possible value of the secret, even if the system would behave in the same way for most of them. In contrast, we have proposed an automata based technique [3], using Markovian models. One state of a model represents an interval of values of the secret for which the system behaves in the same way, allowing for a much more compact and tractable representation.

We build a Markov chain representing the behavior observed by the attacker, then we hide the states that are not observable by the attacker, obtaining a smaller Markov chain—an *observable reduction*. Then we calculate the correlation between the output the attacker can observe and the behavior dependent on the secret, as it corresponds to the leakage. Since leakage in this case is mutual information, it can be computed by adding the entropy of the observable and secret-dependent views of the system and subtracting the entropy of the behavior depending on both. See [3] for details.

QUAIL (QUantitative Analyzer for Imperative Languages) implements this method. It is the first tool supporting arbitrary-precision quantitative evaluation of information leakage for randomized systems or protocols with secret data, including anonymity and authentication protocols. Systems are specified in a simple imperative modeling language further described on QUAILS website.

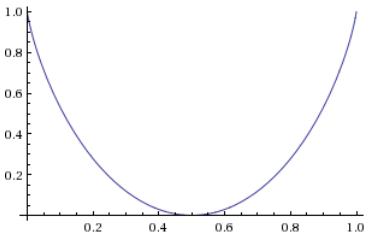


Fig. 1. Bit XOR leakage as a function of $\Pr(r = 1)$

QUAIL performs a white-box leakage analysis assuming that the attacker has knowledge of the system's code but no knowledge of the secret's value, and outputs the result and eventually information about the computation, including the Markov chains computed during the process.

Example. Consider a simple XOR operation example. Variable h stores a 1-bit secret. The protocol generates a random bit r, where $r = 1$ with probability p. It outputs the result of exclusive-or between values of h and r. The attacker knows p and can observe the output, so if $h = r$, but not the values of r or h.

If $p = 0.5$ the attacker cannot infer any information about h, the leakage is zero bits (non-interference). If $p = 0$ or $p = 1$ then she can determine precisely the value of h, and thus the leakage is 1 bit. This can be verified efficiently with language-based tools like APEX [10]. However, QUAIL is the only tool able to precisely compute the leakage for all possible values of p with arbitrary precision. Figure 1 shows that XOR protocol leaks more information as the value of r becomes more deterministic. For instance $p = 0.4$ is safer than $p = 0.8$.

2 QUAIL Implementation

The input model is specifed in QUAIL's imperative language designed to facilitate succinct and direct modeling of protocols, providing features such as arbitrary-size integer variables and arrays, random assignments, `while` and `for` loops, named constants and conditional statements. Figure 2 presents the input code for the bit XOR example.

For a given input code QUAIL builds an annotated Markov chain representing all possible executions of the protocol, then modifies it to encode the protocol when observed by the attacker whose aim is to discover the protocol's secret data. Finally, QUAIL extracts a model of the observable and secret-dependent behavior of the system, and computes the correlation between them, which is equivalent to the amount of bits of the secret that the attacker can infer by observing the system. We now discuss QUAIL implementation following the five steps of the method proposed in [3]:

Step 1: Preprocessing. QUAIL translates the input code into a simplified internal language. It rewrites conditional statements and loops (`if`, `for` and `while`) to conditional jumps (`if-goto`) and substitutes values for named constant references.

Step 2: Probabilistic symbolic execution. QUAIL performs a symbolic forward execution of the input program constructing its semantics as a finite Markov chain (a fully probabilistic transition system) with a single starting state. To this end, QUAIL needs to know the attacker's probability distributions over the secret variables. For each conditional branch, we compute the conditional probability of the guard being satisfied given the values of the public variables and the probability distributions over the secret variables. Then QUAIL generates two successor states, one for the case in which the guard is satisfied and one when not satisfied. This is the most time-consuming step, so QUAIL uses an on-the-fly optimization to avoid building internal states that would be removed in the next step. For instance, it does not generate new states for assignments to a non-observable public variable. Instead it changes the value of the variable in the current state.

Step 3: State hiding and model reduction. To represent what the attacker can examine, QUAIL reduces the Markov chain model by iteratively hiding all unobservable states. For the standard attacker, these are all the internal states, i.e. all the states except the initial and the output states. A state is hidden by creating transitions from its predecessors to its successors and removing it from the model. This operation normally eliminates more than 90% of the states of the Markov chain model, building its *observable reduction*. This operation also detects non-terminating loops and collapses then in a single non-termination state. States are equipped with a list of their predecessors and successors to quicken this step. An observable reduction looks like a probability distribution from the starting states to the output states, since all other states are hidden.

Step 4: Quotienting. Recall from Sect.1 that we have to quantify the correlation between the observable and secret-dependent views of the system. QUAIL relies on the notion of quotients to represent different views of the system and compute their correlation. A quotient is a Markov chain obtained by merging together states in the observable

```
1  observable int1 l; // bit l is the output
2  public int1 r; // bit r is random
3  secret int1 h; // bit h is the secret
4  random r:=randombit(0.5); // randomize r
5  if (h==r) then // calculate the XOR
6  assign l:=0;
7  else
8  assign l:=1;
9  fi
10 return; //terminate
```

```
1  observable int1 l;
2  public int1 r;
3  secret int1 h;
4  random (r):=randombit(0.5);
5  if ((h)==(r))
6  then goto 8;
7  else goto 10;
8  assign (l):=(0);
9  goto 11;
10 assign (l):=(1);
11 return;
```

Fig. 2. Bit XOR example: input code (on the left) and preprocessed code (on the right)

reduction that give the same value to some of the variables. QUAIL quotients the observable reduction separately three times to build three different views of the system. QUAIL uses the attacker model again to know which states are indistinguishable as they assign the same values to the observable variables. These states are merged in the *attacker's quotient*. Similarly, in the *secret's quotient* states are merged if they have the same possible values for the secret, while in the *joint quotient* states are merged if they both have the same values for the secret and cannot be discriminated by the attacker. Since information about the states' variables is not needed to compute entropy, quotients carry none, reducing time and memory required to compute them.

Step 5: Entropy and leakage computation. The *information leakage* can be computed as the sum of the entropies of the attacker's and secret's quotients minus the entropy of the joint quotient [3]. The three entropy computations are independent and can be parallelized. QUAIL outputs the leakage with the desired amount of significant digits and the running time in milliseconds. If requested, QUAIL plots the Markov chain models using Graphviz.

3 On Using QUAIL

QUAIL is freely available from https://project.inria.fr/quail, including source code, binaries and example files. We demonstrate usage of QUAIL to analyze the bit XOR example. Let bit_xor.quail be the file containing the input shown in Fig. 2. The command

```
quail bit_xor.quail -p 2 -v 0
```

executes QUAIL with precision limited to 2 digits (`-p 2`), suppressing all output except the leakage result (`-v 0`). In response QUAIL generates a file bit_xor.quail.pp with the preprocessed code shown in Fig. 2, analyzes it and finally answers `0.0` showing that in this case the protocol leaks no information (so non-interference). For different probability of the random bit r in line 4 QUAIL obtains a different leakage (cf. Fig. 1). For instance, for $p = 0.8$ the leakage is $\sim$0.27807 bits.

4 Comparison with Other Tools

QUAIL precisely evaluates the value of leakage of the input code. This not only allows proving non-interference (absence of leakage) but also enables comparing relative safety of similar protocols. This is particularly important for protocols that exhibit inherent leakage, such as authentication protocols. For instance, with a simple password

Table 1. QUAIL analysis of the leakage in an authentication program

Password length	2	32	64	500
Leakage	$8.11 \cdot 10^{-1}$	$7.78 \cdot 10^{-9}$	$3.54 \cdot 10^{-18}$	$1.52 \cdot 10^{-148}$

authentication, the user inputs a password and is granted access privilege if the password corresponds to the secret stored in the system. The chance of an attacker guessing a password is always positive (although it depends on the password's length). Also, even if the attacker gets rejected she learns something about the secret—the fact that the attempted value was not correct. QUAIL can quantify the precise leakage as a function of the bit length of the password, as shown in Table 1.

Existing *qualitative* tools can establish whether a protocol is completely secure or not, i.e. whether it respects non-interference. They cannot discriminate protocols that allow acceptable and unacceptable violations of non-interference. APEX [10] is an analyzer for probabilistic programs that can check programs equivalence, while PRISM [14] is a probabilistic model-checker. With these tools authentication protocols will always be flagged as unsafe, and a comparison between them is impossible.

QUAIL can be used also to analyze anonymity protocols, like the grade protocol and the dining cryptographers [6]. The interested reader can find discussion and input code for these examples on the QUAIL website. These protocols provide full anonymity on the condition that some random data is generated with a uniform probability distribution; their effectiveness in these cases can be efficiently verified with the qualitative tools above. If the probability distribution over the random data is not uniform some private data is leaked, and QUAIL is again the only tool that can quantify this leakage. Presently, the models for these protocols tend to grow exponentially, so the analysis becomes time-consuming already for about 6–7 agents.

Qualitative tools and technique are more closely related to QUAIL; we present some of them and discuss the main differences. It is worth noting that most of them either do not work for analyzing probabilistic programs [1,11,13,17] or are based on a channel matrix with an impractical number of lines [4,7].

JPF-QIF [19] is a tool that computes an upper bound of the leakage by estimating the number of possible outputs of the system. JPG-QIF is much less precise than QUAIL, and it is not able for instance to prove that the security of an authentication increases by increasing the password size.

McCamant and Ernst [16] and Newsome, McCamant and Song [18] propose quantitative extensions of taint analysis. This approach, while feasible even for large programs, still does not allow to analyze probabilistic programs, making it unsuitable for security protocols.

Bérard et al. propose a quantification of information leakage based on mutual information, though they name it restrictive probabilistic opacity [2] and do not refer to some of the core papers of the subject, like the works of Clark, Hunt and Malacaria [8,9]. The approach tries to quantify leakage on probabilistic models, and is thus phylosophically close to ours. They compute mutual information as the expected difference between prior and posterior entropy, and since the latter depends on all possible values of the secret we expect that an eventual implementation would be in general very inefficient compared to the QUAIL quotient-based approach.

References

1. Backes, M., Köpf, B., Rybalchenko, A.: Automatic discovery and quantification of information leaks. In: IEEE Symposium on Security and Privacy (2009)
2. Bérard, B., Mullins, J., Sassolas, M.: Quantifying opacity. In: Ciardo, G., Segala, R. (eds.) QEST 2010. IEEE Computer Society (September 2010)
3. Biondi, F., Legay, A., Malacaria, P., Wąsowski, A.: Quantifying information leakage of randomized protocols. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 68–87. Springer, Heidelberg (2013)
4. Chatzikokolakis, K., Chothia, T., Guha, A.: Statistical measurement of information leakage. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 390–404. Springer, Heidelberg (2010)
5. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. In: Montanari, U., Sannella, D., Bruni, R. (eds.) TGC 2006. LNCS, vol. 4661, pp. 281–300. Springer, Heidelberg (2007)
6. Chaum, D.: The dining cryptographers problem: Unconditional sender and recipient untraceability. Journal of Cryptology 1, 65–75 (1988)
7. Chothia, T., Guha, A.: A statistical test for information leaks using continuous mutual information. In: CSF, pp. 177–190. IEEE Computer Society (2011)
8. Clark, D., Hunt, S., Malacaria, P.: Quantitative analysis of the leakage of confidential data. Electr. Notes Theor. Comput. Sci. 59(3), 238–251 (2001)
9. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15 (2007)
10. Kiefer, S., Murawski, A.S., Ouaknine, J., Wachter, B., Worrell, J.: APEX: An analyzer for open probabilistic programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 693–698. Springer, Heidelberg (2012)
11. Klebanov, V.: Precise quantitative information flow analysis using symbolic model counting. In: Martinelli, F., Nielson, F. (eds.) QASA (2012)
12. Köpf, B., Basin, D.A.: An information-theoretic model for adaptive side-channel attacks. In: ACM Conference on Computer and Communications Security (2007)
13. Köpf, B., Mauborgne, L., Ochoa, M.: Automatic quantification of cache side-channels. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 564–580. Springer, Heidelberg (2012)
14. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: Verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
15. Malacaria, P.: Algebraic foundations for information theoretical, probabilistic and guessability measures of information flow. CoRR, abs/1101.3453 (2011)
16. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI. ACM (2008)
17. Mu, C., Clark, D.: A tool: Quantitative analyser for programs. In: QEST. IEEE Computer Society (2011)
18. Newsome, J., McCamant, S., Song, D.: Measuring channel capacity to distinguish undue influence. In: Chong, S., Naumann, D.A. (eds.) PLAS. ACM (2009)
19. Phan, Q.-S., Malacaria, P., Tkachuk, O., Pasareanu, C.S.: Symbolic quantitative information flow. ACM SIGSOFT Software Engineering Notes 37(6), 1–5 (2012)
20. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
21. Smith, G.: On the foundations of quantitative information flow. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 288–302. Springer, Heidelberg (2009)

Lengths May Break Privacy – Or How to Check for Equivalences with Length*

Vincent Cheval[1], Véronique Cortier[2], and Antoine Plet[2]

[1] School of Computer Science, University of Birmingham
[2] LORIA, CNRS, France

Abstract. Security protocols have been successfully analyzed using symbolic models, where messages are represented by terms and protocols by processes. Privacy properties like anonymity or untraceability are typically expressed as equivalence between processes. While some decision procedures have been proposed for automatically deciding process equivalence, all existing approaches abstract away the information an attacker may get when observing the length of messages.

In this paper, we study process equivalence with length tests. We first show that, in the static case, almost all existing decidability results (for static equivalence) can be extended to cope with length tests. In the active case, we prove decidability of trace equivalence with length tests, for a bounded number of sessions and for standard primitives. Our result relies on a previous decidability result from Cheval *et al* [15] (without length tests). Our procedure has been implemented and we have discovered a new flaw against privacy in the biometric passport protocol.

1 Introduction

Privacy is an important concern in our today's life where many documents and transactions are digital. For example, we are usually carrying RFIDs cards (for ground transportation, access to office buildings, for opening modern cars, etc.). Due to these cards, malicious users may (attempt to) track us or learn more about us. For instance, the biometric passport contains a chip that stores sensitive information such as birth date, nationality, picture, fingerprints, and also iris characteristics. In order to protect passport holders privacy, the application (or protocol) deployed on biometric passports is designed to achieve authentication without revealing any information to a third party (data is sent encrypted). However, it is well known that designing security protocols is error prone. For example, it was possible to track French citizens due to an additional error message introduced in French passports [5]. Symbolic models have been very successful for analyzing security protocols. Several automatic tools have been designed such as ProVerif [10], Avispa [6], *etc.*. They are very effective to detect flaws or prove

* This work has been partially supported by the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement no 258865, project ProSecure and project JCJC VIP n^o 11 JS02 006 01.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 708–723, 2013.

security of many real-case studies (e.g JFK [2], OAuth2.0 [7], *etc.*). However, these tools are, for most of them, dedicated to accessibility properties. While data secrecy or authentication can be easily expressed as accessibility properties, privacy properties are instead stated as indistinguishability (or equivalence) properties: Alice remains anonymous if an attacker cannot distinguish a session with Alice as participant from a session with Bob as participant. The literature on how to decide equivalence of security protocols is much less prolific than for accessibility. Some procedures have been proposed [8,15,12,11] for some classes of cryptographic primitives, not all procedures being guaranteed to terminate. However, none of these results take into account the fact that an attacker can always observe the length of a message. For example, even if k is a secret key, the cyphertext $\{n\}_k$ corresponding to the encryption of a random number n by the key k can always be distinguished from the cyphertext $\{n, n\}_k$ corresponding to the encryption of a random number n repeated twice by the key k. This is simply due to the fact that $\{n, n\}_k$ is longer than $\{n\}_k$. These two messages would be considered as indistinguishable in all previous mentioned symbolic approaches. The fact that encryption reveals the length of the underling plaintext is a well-identified issue in cryptography. Therefore and not surprisingly, introducing a length function becomes necessary in symbolic models when proving that symbolic process equivalence implies cryptographic indistinguishability [16].

Our contributions. In this paper, we consider an equivalence notion that takes into account the information leaked by the length of a message. More precisely, we equip the term algebra T with a length function $\ell : T \mapsto \mathbb{R}^+$ that associates a non negative real number to any term and we let the attacker compare the length of any two messages he can construct. As usual, the properties of the cryptographic primitives are modeled through an equational theory. For example, the equation $\mathsf{sdec}(\mathsf{senc}(m, k), k)$ models the fact that decrypting with a key k a message m (symmetrically) encrypted by k yields the message m in clear. The goal of our paper is to study the decidability of equivalence with length tests.

The simplest case is the so-called *static case*, where an attacker can only observe protocol executions. Two sequences of messages are *statically equivalent* if an attacker cannot see the difference between them. For example, the two messages $\{0\}_k$ and $\{1\}_k$ are distinct but cannot be distinguished by an attacker unless he knows the key k. We show how most existing decidability results for static equivalence can be extended to length tests. We simply require the length function to be homomorphic, that is, the length $\ell(M)$ of a term $M = f(M_1, \ldots, M_k)$ is a function of f and the lengths of $M_1, \ldots, M_k$. We show that whenever static equivalence is decidable for some equational theory E then static equivalence remains decidable when adding length tests. The result requires a simple hypothesis called SET-stability that is satisfied by most equational theories that have been showed decidable for static equivalence. As an application, we deduce decidability of static equivalence for many primitives, including symmetric and asymmetric encryption, signatures, hash, blind signatures, exclusive or, *etc.*

The *active case*, where an attacker can freely interact with the protocol, is of course more involved. Even without the introduction of a length function,

there are very few decidability results [17,15]. Starting from the decision procedure developed in [15], we show how to deal with length functions for the standard cryptographic primitives (symmetric and asymmetric encryption, signatures, hash, and concatenation). Like for the static case, our result is actually very modular. In order to check whether two protocols P and Q are in trace equivalence with length tests, it is sufficient to first run the procedure of [15], checking whether two protocols P and Q are in trace equivalence without length tests. It is then sufficient to check for equalities of the polynomials we derive from the processes that appear in the final states of the procedure of [15]. As such, we provide a decision procedure for the two following problems: (1) Given two processes P and Q and a length function ℓ, are P and Q in trace equivalence with length tests (w.r.t. the length function ℓ)? (2) Given two processes P and Q, does there exist a length function ℓ such that P and Q in not trace equivalence with length tests (w.r.t. the length function ℓ)? From a practical point of view, this amounts into deciding whether there exists an implementation of the primitives (that would meet some particular length property) such that an attacker could distinguish between P and Q, leading to a privacy attack. We have implemented our decision procedure for trace equivalence with length tests as an extension of the APTE tool developped for [15]. As an application, we study the biometric passport [1] and discover a new flaw. We show that an attacker can break privacy by observing messages of different lengths depending on which passport is used, therefore discovering who between Alice or Bob is currently using her/his passport.

Related work. Existing decision procedures for trace equivalence do not consider length tests. [15] shows that trace equivalence is decidable for finitely many sessions and for a fixed term algebra (encryption, signatures, hash, ...). A procedure for a more flexible term algebra is provided in [12] but is not guaranteed to terminate. Building on [8], it has been shown that trace equivalence can be decided for any convergent subterm equational theories, for protocols with no else branches [17]. The tool ProVerif [10,11] is also able to check for equivalence but is again not guaranteed to terminate (and prove an equivalence that is sometimes too strong). One of the only symbolic models that introduce a length function is the model developed in [16] for proving that symbolic process equivalence implies cryptographic indistinguishability. However, [16] does not discuss any decision procedure for process equivalence.

2 Preliminaries

A key ingredient of formal models for security protocols is the representation of messages by *terms*. This section is devoted to the definitions of terms and two key notions of knowledge for the attacker: deduction and static equivalence.

2.1 Terms

Given a *signature* $\mathcal{F}$ (*i.e.* a finite set of function symbols, with a given arity), an infinite set of *names* $\mathcal{N}$, and an infinite set of variables $\mathcal{X}$, the set of terms

$T(\mathcal{F}, \mathcal{N}, \mathcal{X})$ is defined as the union of names $\mathcal{N}$, variables $\mathcal{X}$, and function symbols of $\mathcal{F}$ applied to other terms. A term is said to be ground if it contains no variable. $\tilde{n}$ denotes a set of names. The set of names of a term M is denoted by $fnames(M)$. Substitutions are replacement of variables by terms and are denoted by $\theta = \{M_1/x_1, \ldots, M_k/x_k\}$. The application of a substitution θ to a term M is defined as usual and is denoted $M\theta$. A *context* C is a term with holes. Given terms $M_1, \ldots, M_k$, the term $C[M_1, \ldots, M_k]$ may be denoted $C[\tilde{M_i}]$.

Example 1. A signature for modelling the standard cryptographic primitives (symmetric and asymmetric encryption, concatenation, signatures, and hash) is $\mathcal{F}_{\mathsf{stand}} = \mathcal{F}_c \cup \mathcal{F}_d$ where $\mathcal{F}_c$ and $\mathcal{F}_d$ are defined as follows (the second argument being the arity):

$$\mathcal{F}_c = \{\mathsf{senc}/2,\ \mathsf{aenc}/2,\ \mathsf{pk}/1,\ \mathsf{sign}/2,\ \mathsf{vk}/1,\ \langle\,\rangle/2,\ \mathsf{h}/1\}$$
$$\mathcal{F}_d = \{\mathsf{sdec}/2,\ \mathsf{adec}/2,\ \mathsf{check}/2,\ \mathsf{proj}_1/1,\ \mathsf{proj}_2/1\}.$$

The function aenc (resp. senc) represents asymmetric encryption (resp. symmetric encryption) with corresponding decryption function adec (resp. sdec) and public key pk. Concatenation is represented by $\langle\,\rangle$ with associated projectors proj_1 and proj_2. Signature is modeled by the function sign with corresponding validity check check and verification key vk. h represents the hash function.

The properties of the cryptographic primitives (e.g. decrypting an encrypted message yields the message in clear) are expressed through equations. Formally, we equip the term algebra with an *equational theory*, that is, an equivalence relation on terms which is closed under substitutions for variables and names. We write $M =_E N$ when the terms M and N are equivalent modulo E. Equational theories can be used to specify a large variety of cryptographic primitives, from the standard cryptographic primitives of Example 1 to exclusive or (XOR), blind signatures, homomorphic encryption, trapdoor-commitment or Diffie-Hellman. We provide below a theory for the standard primitives and for XOR. More examples of equational theories can be found in [3,4].

Example 2. Continuing Example 1, the equational theory E_{stand} for the standard primitives is defined by the equations:

$$\mathsf{sdec}(\mathsf{senc}(x, y), y) = x \quad (1)$$
$$\mathsf{adec}(\mathsf{aenc}(x, \mathsf{pk}(y)), y) = x \quad (2)$$
$$\mathsf{check}(\mathsf{sign}(x, y), \mathsf{vk}(y)) = x \quad (3)$$
$$\mathsf{proj}_1(\langle x, y\rangle) = x \quad (4)$$
$$\mathsf{proj}_2(\langle x, y\rangle) = y \quad (5)$$

Equation 1 models that decrypting an encrypted message $\mathsf{senc}(m, k)$ with the right key k yields the message m in clear. Equation 2 is the asymmetric analog of Equation 1. Similarly, Equations 4 and 5 model the first and second projections for concatenation. There are various ways for modeling signature. Here, Equation 3 models actually two properties. First, the validity of a signature $\mathsf{sign}(m, k)$ given the verification key $\mathsf{vk}(k)$ can be checked by applying the test function check. Second, the underlying message m under signature can be retrieved (as it is often the case in symbolic models). This is because we assume

that a signature $\mathsf{sign}(m, k)$, which represents the digital signature itself, is always sent together with the corresponding message m.

Example 3. The theory of XOR $E_{\oplus}$, is based on the signature $\Sigma = \{\oplus/2, 0/0\}$ and the equations:

$$(x \oplus y) \oplus z = x \oplus (y \oplus z) \qquad x \oplus x = 0$$
$$x \oplus y = y \oplus x \qquad x \oplus 0 = x$$

The two left equations model the fact that the function $\oplus$ is *associative* and *commutative*. The right equations model the fact that XORing twice the same element yields the neutral element 0.

A function symbol $+$ is said to be AC (associative and commutative) if it satisfies the two equations $(x + y) + z = x + (y + z)$ and $x + y = y + x$. For example, the symbol $\oplus$ is an AC-symbol of the theory $E_{\oplus}$. Given an equational theory E, we write $M =_{AC} N$ if M and N are equal modulo the associativity and commutativity of their AC-symbols.

2.2 Deduction and Static Equivalence

During protocol executions, the attacker learns sequences of messages $M_1, \ldots, M_k$ where some names are initially unknown to him. This is modeled by defining a *frame* ϕ to be an expression of the form

$$\phi = \nu\tilde{n}\{M_1/x_1, \ldots, M_k/x_k\}$$

where $\tilde{n}$ is a set of names (representing the secret names) and the M_i are terms. A frame is *ground* is all its terms are ground. The *domain* of the frame ϕ is $\mathrm{dom}(\phi) = \{x_1, \ldots, x_n\}$.

The first basic notion when modeling the attacker is the notion of *deduction*. It captures what an attacker can built from a frame ϕ. Intuitively, the attacker knows all the terms of ϕ and can apply any function to them.

Definition 1 (deduction). *Given an equational theory E and a frame $\phi = \nu\tilde{n}\sigma$, a ground term N is* deducible *from ϕ, denoted $\phi \vdash N$, if there is a* free *term M (i.e. $fnames(M) \cap \tilde{n} = \emptyset$), such that $M\sigma =_E N$. The term M is called a* recipe *of N.*

Example 4. Consider $\phi_1 = \nu n, k, k'\{k/x_1, \mathsf{senc}(\langle n, n\rangle, k)/x_2, \mathsf{senc}(n, k')/x_3\}$. Then $\phi_1 \vdash k$, $\phi_1 \vdash n$, but $\phi_1 \nvdash k'$. A recipe for k is x_1 while a recipe for n is $\mathsf{proj}_1(\mathsf{sdec}(x_2, x_1))$. Another possible recipe of n is $\mathsf{proj}_2(\mathsf{sdec}(x_2, x_1))$.

As mentioned in the introduction, the confidentiality of a vote v or the anonymity of an agent a cannot be defined as the non deducibility of v or a. Indeed, both are in general public values and are thus always deducible. Instead, the standard approach consists in defining privacy based on an indistinguishability notion: an execution with a should indistinguishable from an execution with b. Indistinguishability of sequences of terms is formally defined as static equivalence.

Definition 2 (static equivalence). *Two frames* $\phi_1 = \nu\tilde{n_1}\sigma_1$ *and* $\phi_2 = \nu\tilde{n_2}\sigma_2$ *are* statically equivalent*, denoted* $\phi_1 \sim \phi_2$*, if and only if for all terms* M, N *such that* $(fn(M) \cup fn(N)) \cap (\tilde{n_1} \cup \tilde{n_2}) = \emptyset$*,*

$$(M\sigma_1 =_E N\sigma_1) \Leftrightarrow (M\sigma_2 =_E N\sigma_2).$$

Example 5. Let $\phi_2 = \nu n, n', k, k'\{k/x_1, \mathsf{senc}(\langle n', n\rangle, k)/x_2, \mathsf{senc}(n, k')/x_3\}$ and $\phi_3 = \nu n, k, k'\{k/x_1, \mathsf{senc}(\langle n, n\rangle, k)/x_2, \mathsf{senc}(\langle n, n\rangle, k')/x_3\}$. ϕ_1 is defined in Example 4. Then $\phi_1 \not\sim \phi_2$ since $\mathsf{proj}_1(\mathsf{sdec}(x_2, x_1)) = \mathsf{proj}_2(\mathsf{sdec}(x_2, x_1))$ is true in ϕ_1 but not in ϕ_2. Intuitively, an attacker may distinguish between ϕ_1 and ϕ_2 by decrypting the second message and noticing that the two components are equal for ϕ_1 while they differ for ϕ_2. Conversely, we have $\phi_1 \sim \phi_3$.

2.3 Rewrite Systems

To decide deduction and static equivalence, it is often convenient to reason with a rewrite system instead of an equational theory. A *rewrite system* $\mathcal{R}$ is a set of rewrite rules $l \to r$ (where l and r are terms) that is closed by substitution and context. Formally a term u can be rewritten in v, denoted by $u \to_{\mathcal{R}} v$ if there exists $l \to r \in \mathcal{R}$, a substitution θ, and a position p of u such that $u|_p = l\theta$ and $v = u[r\theta]_p$. The transitive and reflexive cloture of $\to_{\mathcal{R}}$ is denoted $\to^*_{\mathcal{R}}$. We write $\to$ instead of $\to_{\mathcal{R}}$ when $\mathcal{R}$ is clear from the context.

Definition 3 (convergent). *A rewrite system* $\mathcal{R}$ *is* convergent *if it is:*

- terminating*: there is no infinite sequence* $u_1 \to u_2 \to \cdots \to u_n \to \cdots$
- confluent*: for every terms* u, u_1, u_2 *such that* $u \to u_1$ *and* $u \to u_2$*, there exists* v *such that* $u_1 \to^* v$ *and* $u_2 \to^* v$*.*

For a convergent rewrite system, a term t *has a unique* normal form $t{\downarrow}$ *such that* $t \to^* t{\downarrow}$ *and* $t{\downarrow}$ *has no successor.*

An equational theory E *is* convergent *if there exists a finite convergent rewrite system* $\mathcal{R}$ *such that for any two terms* u, v*, we have* $u =_E v$ *if and only if* $u{\downarrow} = v{\downarrow}$*.*

For example, the theory E_{stand} defined in Example 2 is convergent. Its associated finite convergent rewrite system is obtained by orienting the equations from left to right. Conversely, the theory $E_{\oplus}$ defined in Example 3 is not convergent due the equations of associativity and commutativity. Since many equational theories modeling cryptographic primitives do have associative and commutative symbols, we define *rewriting modulo* AC as $M \to_{AC} N$ if there is a term M' such that $M =_{AC} M'$ and $M' \to N$. AC-convergence can then be defined similarly to convergence.

Definition 4 (AC-convergent). *A rewrite system* $\mathcal{R}$ *is* AC-convergent *if it is:*

- AC-terminating*: there is no infinite sequence* $u_1 \to_{AC} \cdots \to_{AC} u_n \to_{AC} \cdots$
- AC-confluent*: for every terms* u, u_1, u_2 *such that* $u \to_{AC} u_1$ *and* $u \to_{AC} u_2$*, there exists* v *such that* $u_1 \to^*_{AC} v$ *and* $u_2 \to^*_{AC} v$*.*

For a AC-convergent rewrite system, a term t has a unique set of normal forms $t{\downarrow_{AC}} = \{t' \mid t \to^* t' \text{ and } t' \text{ has no successor}\}$. *For any $u, v \in t{\downarrow_{AC}}$, $u =_{AC} v$.*

An equational theory E is AC-convergent *if there exists a finite AC-convergent rewrite system $\mathcal{R}$ such that for any two terms u, v, we have $u =_E v$ if and only if $u{\downarrow_{AC}} = v{\downarrow_{AC}}$.*

For example, the theory $E_{\oplus}$ defined Example 3 is AC-convergent. Its associated finite AC-convergent rewrite system is obtained by orienting the two right equations from left to right. Of course, any convergent theory is AC-convergent. Most, if not all, equational theories for cryptographic primitives are convergent or at least AC-convergent. So in what follows, we only consider AC-convergent theories.

3 Length Equivalence - Static Case

While many decidability results have been provided for deduction and static equivalence, for various theories, none of them study the leak induced by the length of messages. In this section, we provide a definition for length functions and we study how to extend existing decidability results to length functions.

3.1 Length Function

A *length function* is simply a function $\ell : T(\mathcal{F}, \mathcal{N}, \mathcal{X}) \to \mathbb{R}^+$ that associates non-negative real numbers to terms. A meaningful length function should associate the same length to terms that are equal modulo the equational theory. Since we consider AC-convergent theories, we assume that the length of a term t is evaluated by an auxiliary function applied once t is in normal form. Moreover, the size of a term $f(M_1, \ldots, M_k)$ is typically a function that depends on f and the length of $M_1 \ldots, M_k$. This class of length functions is called *normalized length functions*.

Definition 5 (Normalized length function). *Let $T(\mathcal{F}, \mathcal{N}, \mathcal{X})$ be a term algebra and E be an AC-convergent equational theory. A length function ℓ is a* normalized length function *if there exists a function $\ell_{\mathsf{aux}} : T(\mathcal{F}, \mathcal{N}, \mathcal{X}) \to \mathbb{R}^+$ (called auxiliary length function) such that the following properties hold:*

1. *ℓ_{aux} is a morphism, that is, for every function symbol f of arity k, there exists a function $\ell_f : {\mathbb{R}^+}^k \to \mathbb{R}^+$ s.t. for all terms $M_1, \ldots, M_k$ $\ell_{\mathsf{aux}}(f(M_1, \ldots, M_k)) = \ell_f(\ell_{\mathsf{aux}}(M_1), \ldots, \ell_{\mathsf{aux}}(M_k))$*
2. *ℓ_{aux} is stable modulo AC: $\ell_{\mathsf{aux}}(M) = \ell_{\mathsf{aux}}(N)$ for all M, N s.t. $M =_{AC} N$.*
3. *ℓ_{aux} decreases with rewriting: $\ell_{\mathsf{aux}}(M) \geq \ell_{\mathsf{aux}}(N)$ for all M, N s.t. $M \to_{AC} N$.*
4. *ℓ coincides with ℓ_{aux} on normal forms: $\ell(M) = \ell_{\mathsf{aux}}(M{\downarrow_{AC}})$ where $\ell_{\mathsf{aux}}(M{\downarrow_{AC}})$ is defined to be $\ell_{\mathsf{aux}}(N)$ for any $N \in M{\downarrow_{AC}}$.*
5. *For any $r \in \mathbb{R}^+$, the set $\{n \in \mathcal{N} \mid \ell(n) = r\}$ is either infinite or empty. A name should not be particularized by its length.*

Note that item 5 implies in particular that $\ell_{\mathsf{aux}}(M) = \ell_{\mathsf{aux}}(M\sigma)$ for any σ that replaces the names of M by names of equal length (i.e. such that $\ell_{\mathsf{aux}}(\sigma(n) = \ell_{\mathsf{aux}}(n))$. Indeed, the length should not depend of the choice of names.

Example 6. A natural length function for the standard primitives defined in Example 2 is ℓ_{stand} induced by the following auxiliary length function ℓ_{aux}:

$$\begin{aligned}
\ell_{\mathsf{aux}}(n) &= 1 && n \in \mathcal{N}\\
\ell_{\mathsf{aux}}(\mathsf{senc}(u,v)) &= \ell_{\mathsf{aux}}(u) + \ell_{\mathsf{aux}}(v)\\
\ell_{\mathsf{aux}}(\langle u,v\rangle)) &= 1 + \ell_{\mathsf{aux}}(u) + \ell_{\mathsf{aux}}(v)\\
\ell_{\mathsf{aux}}(\mathsf{aenc}(u,v)) &= 2 + \ell_{\mathsf{aux}}(u) + \ell_{\mathsf{aux}}(v)\\
\ell_{\mathsf{aux}}(\mathsf{sign}(u,v)) &= 3 + \ell_{\mathsf{aux}}(u) + \ell_{\mathsf{aux}}(v)\\
\ell_{\mathsf{aux}}(f(u,v)) &= 1 + \ell_{\mathsf{aux}}(u) + \ell_{\mathsf{aux}}(v) && f \in \{\mathsf{sdec}, \mathsf{adec}, \mathsf{check}\}\\
\ell_{\mathsf{aux}}(f(u)) &= 1 + \ell_{\mathsf{aux}}(u) && f \in \{\mathsf{proj}_1, \mathsf{proj}_2\}
\end{aligned}$$

Then the length of a term M is simply the auxiliary length of its normal form: $\ell(M) = \ell_{\mathsf{aux}}(M{\downarrow})$ and ℓ is a normalized length function. Note that the constants 1, 2, 3 are rather arbitrary and ℓ would be a normalized length function for any other choice. The choice of the exact parameters typically depends on the implementation of the primitives.

Example 7. A length function for XOR is $\ell_{\oplus}$, induced by the auxiliary function ℓ_{aux} defined by $\ell_{\mathsf{aux}}(n) = 1$ for n name, $\ell_{\mathsf{aux}}(0) = 0$, and $\ell_{\mathsf{aux}}(u \oplus v) = \max(\ell_{\mathsf{aux}}(u), \ell_{\mathsf{aux}}(v))$. Then $\ell_{\oplus}$ is again a normalized length function.

An attacker may compare the length of messages, which gives him additional power. For example, the frames ϕ_1 and ϕ_3 (defined in Example 5) are statically equivalent. However, in reality, an attacker would notice that the third messages are of different length. In particular, $\ell_{\mathsf{stand}}(\mathsf{senc}(n, k')) = 2$ while $\ell_{\mathsf{stand}}(\mathsf{senc}(\langle n, n\rangle, k')) = 4$ (where ℓ_{stand} has been defined in Example 6).

We extend the notion of static equivalence to take into account the ability of an attacker to check for equality of lengths.

Definition 6 (static equivalence w.r.t. length). *Two frames $\phi_1 = \nu\tilde{n_1}\sigma_1$ and $\phi_2 = \nu\tilde{n_2}\sigma_2$ are* statically equivalent w.r.t. the length function ℓ*, denoted $\phi_1 \sim^{\ell} \phi_2$, if ϕ_1 and ϕ are statically equivalent ($\phi_1 \sim \phi_2$) and for all terms M, N such that $(fn(M) \cup fn(N)) \cap (\tilde{n_1} \cup \tilde{n_2}) = \emptyset$,*

$$(\ell(M\sigma_1) =_E \ell(N\sigma_1)) \Leftrightarrow (\ell(M\sigma_2) =_E \ell(N\sigma_2)).$$

3.2 Decidability

Ideally, we would like to inherit any decidability result that exists for the usual static equivalence $\sim$. We actually need to look deeper in how decidability results are obtained for $\sim$. In many approaches (e.g. [3,9]), decidability of static equivalence is obtained by computing from a frame ϕ, an upper set that symbolically describes the set of all deducible subterms. Here, we generalize this property into SET-stability.

Definition 7 (SET-stable). *An equational theory E is* SET-stable *if for any frame $\phi = \nu\tilde{n}\{M_1/x_1, \ldots, M_k/x_k\}$ there exists a set $\mathsf{SET}(\phi)$ such that:*

- $M_1, \ldots, M_k \in \mathsf{SET}(\phi)$,
- $\forall M \in \mathsf{SET}(\phi), \phi \vdash M$,
- *for any finite set of names $\tilde{n'} \supseteq \tilde{n}$, for every context C_1 such that $fn(C_1) \cap \tilde{n'} = \emptyset$, for all $N_i^1 \in \mathsf{SET}(\phi)$, for all $T \in (C_1[\tilde{N_i^1}]\downarrow)$, there exist a context C_2 such that $fn(C_2) \cap \tilde{n'} = \emptyset$ and terms $N_i^2 \in \mathsf{SET}(\phi)$ such that $T =_{AC} C_2[\tilde{N_i^2}]$.*

We say that E is efficiently SET-stable *if there is an algorithm that computes the set $\mathsf{SET}(\phi)$ being given a frame ϕ and that computes a recipe ζ_M for any $M \in \mathsf{SET}(\phi)$.*

We are now ready to state our first main theorem.

Theorem 1. *Let E be an efficiently SET-stable equational theory and ℓ be a normalized length function. If $\sim_E$ is decidable then $\sim_E^\ell$ is decidable.*

Sketch of proof The algorithm for checking for $\sim_E^\ell$ works as follows. Given two frames $\phi_1 = \nu\tilde{n}\sigma_1$ and $\phi_2 = \nu\tilde{n}\sigma_2$,

- check whether $\phi_1 \sim_E \phi_2$
- compute $\mathsf{SET}(\phi_1)$ and $\mathsf{SET}(\phi_2)$;
- for any $M \in \mathsf{SET}(\phi_1)$, compute its corresponding recipe ζ_M and check whether $\ell(\zeta_M\sigma_2) = \ell(M)$;
- symmetrically, for any $M \in \mathsf{SET}(\phi_2)$, compute its corresponding recipe ζ_M and check whether $\ell(\zeta_M\sigma_1) = \ell(M)$;
- return true if all checks succeeded and false otherwise.

The algorithm returns true if $\phi_1 \sim_E^\ell \phi_2$. Indeed, for any $M \in \mathsf{SET}(\phi_1)$, $\ell(\zeta_M\sigma_1) = \ell(M) = \ell(M_0)$ where M_0 is length-preserving renaming of $M\downarrow$ with free names only. $\ell(\zeta_M\sigma_1) = \ell(M_0\sigma_1)$ implies $\ell(\zeta_M\sigma_2) = \ell(M_0\sigma_2) = \ell(M_0) = \ell(M)$.

The converse implication is more involved and makes use of the properties of the sets $\mathsf{SET}(\phi_1)$ and $\mathsf{SET}(\phi_2)$. □

Applying Theorem 1 we can deduce the decidability of $\sim_E^\ell$ for any theory E described in [3], e.g. theories for the standard primitives, for XOR, for pure AC, for blind signatures, homomorphic encryption, addition, *etc.* More generally, we can infer decidability for any *locally stable* theories, as defined in [3]. Intuitively, locally-stability is similar to SET-stability except that only small contexts are considered. Locally-stability is easier to check than SET-stability and has been shown to imply SET-stability in [3].

Corollary 1. *Let E be a locally-stable equational theory as defined in [3]. Let ℓ be a normalized length function. If $\sim_E$ is decidable then $\sim_E^\ell$ is decidable.*

4 Length Equivalence - Active Case

We now study length equivalence in the active case, that is when an attacker may fully interact with the protocol under study. We first define our process algebra, in the spirit of the applied-pi calculus [4].

4.1 Syntax

We consider $\mathcal{F}_d$ as defined in Example 1 and $\mathcal{F}'_c \supseteq \mathcal{F}_c$. We let $\mathcal{F}'_c$ contain more primitives than $\mathcal{F}_c$, to allow for constants or free primitives such as mac. We consider the fixed equational theory E_{stand} as defined in Example 2. Orienting the equations of E_{stand} from left to right yields a convergent rewrite system.

The *constructor terms*, resp. *ground constructor terms*, are those in $\mathcal{T}(\mathcal{F}'_c, \mathcal{N} \cup \mathcal{X})$, resp. in $\mathcal{T}(\mathcal{F}'_c, \mathcal{N})$. A ground term u is called a *message*, denoted $\mathsf{Message}(u)$, if $v{\downarrow}$ is a constructor term for all $v \in st(u)$. For instance, the terms $\mathsf{sdec}(a, b)$, $\mathsf{proj}_1(\langle a, \mathsf{sdec}(a, b)\rangle)$, and $\mathsf{proj}_1(a)$ are not messages. Intuitively, we view terms as modus operandi to compute bitstrings where we use the call-by-value evaluation strategy.

The grammar of our *plain processes* is defined as follows:

$$P, Q := 0 \mid (P \,|\, Q) \mid P + Q \mid \mathsf{in}(u, x).P \mid \mathsf{out}(u, v).P \mid \mathsf{if}\ u_1 = u_2\ \mathsf{then}\ P\ \mathsf{else}\ Q$$

where u_1, u_2, u, v are terms, and x is a variable of $\mathcal{X}$. Our calculus contains parallel composition $P \mid Q$, choice $P+Q$, tests, input $\mathsf{in}(u, x).P$, and output $\mathsf{out}(u, v)$. Since we do not consider restriction, private names can simply be specified before hand so there is no need for name restriction. Trivial else branches may be omitted.

Definition 8 (process). *A* process *is a triple* $(\mathcal{E}; \mathcal{P}; \Phi)$ *where:*

- $\mathcal{E}$ *is a set of names that represents the private names of* $\mathcal{P}$*;*
- Φ *is a ground frame with domain included in* $\mathcal{AX}$*. It represents the messages available to the attacker;*
- $\mathcal{P}$ *is a multiset of* closed plain processes*.*

4.2 Semantics

The semantics for processes is defined as usual. Due to space limitations, we only provide two illustrative rules (see [13] or the appendix for the full definition).

$$(\mathcal{E}; \{\mathsf{in}(u, x).Q\} \uplus \mathcal{P}; \Phi) \xrightarrow{in(N,M)} (\mathcal{E}; \{Q\{x \mapsto t\}\} \uplus \mathcal{P}; \Phi) \qquad (\text{IN}_c)$$
$$\text{if } M\Phi = t,\ fvars(M, N) \subseteq \mathrm{dom}(\Phi),\ fnames(M, N) \cap \mathcal{E} = \emptyset$$
$$N\Phi{\downarrow} = u{\downarrow},\ \mathsf{Message}(M\Phi),\ \mathsf{Message}(N\Phi),\ \text{and}\ \mathsf{Message}(u)$$

$$(\mathcal{E}; \{\mathsf{out}(u, t).Q\} \uplus \mathcal{P}; \Phi) \xrightarrow{\nu ax_n.out(M, ax_n)} (\mathcal{E}; \{Q\} \uplus \mathcal{P}; \Phi \cup \{ax_n \rhd t\}) \ (\text{OUT}_c)$$
$$\text{if } M\Phi{\downarrow} = u{\downarrow},\ \mathsf{Message}(u),\ fvars(M) \subseteq \mathrm{dom}(\Phi),\ fnames(M) \cap \mathcal{E} = \emptyset$$
$$\mathsf{Message}(M\Phi),\ \mathsf{Message}(t)\ \text{and}\ ax_n \in \mathcal{AX},\ n = |\Phi| + 1$$

where u, v, t are ground terms, and x is a variable. The $\xrightarrow{w}$ relation is then defined as usual as the reflexive and transitive closure of $\rightarrow$, where w is the concatenation of all non silent actions.

The set of traces of a process $A = (\mathcal{E}; \mathcal{P}_1; \Phi_1)$ is the set of the possible sequences of actions together with the resulting frame.

$$\mathsf{trace}(A) = \{(s, \nu\mathcal{E}.\Phi_2) \mid (\mathcal{E}; \mathcal{P}_1; \Phi_1) \stackrel{s}{\Rightarrow} (\mathcal{E}; \mathcal{P}_2; \Phi_2) \text{ for some } \mathcal{P}_2, \Phi_2\}$$

4.3 Equivalence

Some terms such as $\mathsf{sdec}(\langle a, b\rangle, k)$ or $\mathsf{sdec}(\mathsf{senc}(a, k'), k)$ do not corresponding to actual messages since the corresponding computation would typically fail and return an error message. It would not make sense to compare the length of such decoy messages. We therefore adapt the notion of static equivalence in order to compare only lengths of terms that correspond to actual messages.

Definition 9. *Let $\mathcal{E}$ a set of private names. Let Φ and Φ' two frames. We say that $\nu\mathcal{E}.\Phi$ and $\nu\mathcal{E}.\Phi'$ are statically equivalent w.r.t. a length function ℓ, written $\nu\mathcal{E}.\Phi \sim_c^\ell \nu\mathcal{E}.\Phi'$, when $\mathrm{dom}(\Phi) = \mathrm{dom}(\Phi')$ and when for all terms M, N such that $fvars(M, N) \subseteq \mathrm{dom}(\Phi)$ and $fnames(M, N) \cap \mathcal{E} = \emptyset$, we have:*

- $\mathsf{Message}(M\Phi)$ *if and only if* $\mathsf{Message}(M\Phi')$
- *if* $\mathsf{Message}(M\Phi)$ *and* $\mathsf{Message}(N\Phi)$ *then*
  - $M\Phi{\downarrow} = N\Phi{\downarrow}$ *if and only* $M\Phi'{\downarrow} = N\Phi'{\downarrow}$*; and*
  - $\ell(M\Phi{\downarrow}) = \ell(N\Phi{\downarrow})$ *if and only if* $\ell(M\Phi'{\downarrow}) = \ell(N\Phi'{\downarrow})$.

Two processes A and B are in trace equivalence if any sequence of actions of A can be matched by the same sequence of actions in B such that the resulting frames are statically equivalent.

Definition 10 (trace equivalence w.r.t. length $\approx^\ell$). *Let A and B be processes with the same set of private names $\mathcal{E}$. $A \sqsubseteq^\ell B$ if for every $(s, \nu\mathcal{E}.\Phi) \in \mathsf{trace}_c(A)$, there exists $(s, \nu\mathcal{E}.\Phi') \in \mathsf{trace}(B)$ such that $\nu\mathcal{E}.\Phi \sim_c^\ell \nu\mathcal{E}.\Phi'$.*

Two closed processes A and B are trace equivalent w.r.t. the length function *ℓ, denoted by $A \approx^\ell B$, if $A \sqsubseteq^\ell B$ and $B \sqsubseteq^\ell A$.*

The length functions associated to standard primitives usually follow a simple pattern (see e.g. Example 6). We focus on *linear* length functions, that have been proved sound w.r.t. symbolic models [16]. A linear function is a function ℓ such that for any $f \in \mathcal{F}'_c$, $\ell(f(t_1, \ldots, t_n)) = l_f(\ell(t_1), \ldots, \ell(t_n))$ where $l_f(x_1, \ldots, x_n) = \beta^f + \sum_{i=1}^n \alpha_i^f x_i$ for some $\alpha_1^f, \ldots, \alpha_n^f, \beta^f \in \mathbb{R}^+$. Moreover, we assume that hashed messages are of fixed size: $\ell(\mathsf{h}(t)) = \ell(n)$ for any term t and name n. Finally, we assume that the size of a pairing is at least the size of its arguments. Our second main contribution is a decision procedure for trace equivalence w.r.t. length.

Theorem 2. *Let ℓ be a linear length function. The problem of trace equivalence w.r.t. ℓ is decidable.*

Even if two processes are in trace equivalence for some length function, they may not be in trace equivalence for another one. Choosing the appropriate length function may be tricky since the "right" parameters depend on the implementation of the primitives. We can actually decide a stronger problem: the existence of a length function that would compromise trace equivalence.

Theorem 3. *The following problem is decidable:*

Entry: *two closed processes A and B*
Question: *does there exist a linear length function ℓ such that $A \not\approx^\ell B$?*

For both theorems, the decision procedure builds upon the decision procedure developed in [15] for trace equivalence (without length). Given two closed processes A and B, our procedure roughly works as follows.

1. We first apply the procedure of [15] to A and B.
2. If $A \not\approx B$ (A and B are not in trace equivalence) then clearly $A \not\approx^{\ell} B$ for any length function ℓ.
3. Otherwise, if $A \approx B$, we look deeper at the output of the procedure of [15]. It ends up with two trees (one for each process), which leaves are sets of "constraint systems" $\mathcal{C}$ that define a parametrized frame $\Phi(\mathcal{C})$. We can associate polynomials to each frame as follows. Given a term u with parameters $param(u)$, we define its *associated polynomial* $P_u \in \mathbb{Z}[param(u)]$ by $P_n = \ell(n)$ for n a name, $P_x = x$ for x a parameter and $P_{f(u_1,\ldots,u_k)} = \ell_f(P_{u_1}, \ldots, P_{u_k})$ otherwise.

Then the sequence of polynomials associated to a frame $\Phi = \{\xi_1 \rhd u_1, \ldots, \xi_n \rhd u_n\}$ is $P_\Phi = P_{u_1}, \ldots, P_{u_n}$.

We can show that $A \approx^{\ell} B$ if and only if, for any set Σ_1 of constraint system that appears as leaf in the tree associated to A, its corresponding set Σ_2 of constraint system in the tree associated to B is such that

$$\{P_{\Phi(\mathcal{C})} \mid \mathcal{C} \in \Sigma_1\} = \{P_{\Phi(\mathcal{C})} \mid \mathcal{C} \in \Sigma_2\}.$$

Therefore, checking for trace equivalence for a particular linear length function ℓ (Theorem 2) amounts into checking for equality of sets of polynomials. Checking whether there exists a linear length function ℓ such that an attacker can distinguish between A and B (Theorem 3) amounts into checking for equality of sets of parametrized polynomials, which in turn amounts again into checking for equality of polynomials (since the coefficients of the parametrized polynomials are also polynomials).

Our procedure could be easily extended to non linear length functions, provided that we can solve the corresponding algebraic problem, that is equality of the zeros of the P_u's, when they are not polynomials anymore.

5 Passport

The biometric passport contains an RFID chip that stores sensitive authentication information such as birth date, nationality, picture, fingerprints, and also iris characteristics. The International Civil Aviation Organisation (ICAO) standard specifies the communication protocols that are used to access these information [1]. We have discovered a new attack on anonymity, as soon as the size of the pictures may vary from one user to another one.

5.1 Description of the Passive Authentication Protocol

According to the ICAO standard, a reader (*e.g.* officer at the border) and a passport first establishes key sessions (denoted *ksenc* and *ksmac*) through the Basic Access Control protocol. Once such keys are successfully established, the

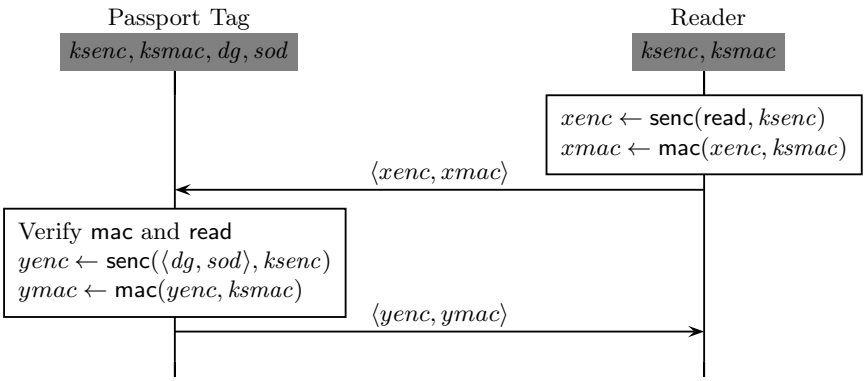


Fig. 1. Passive Authentication protocol (PA)

Passive Authentication protocol (Figure 1) is executed along with other protocols. It establishes a secure communication between the reader and the passport, which sends the (sensitive) authentication information such as the name, date of birth, nationality, and pictures. This information is organised in data groups (dg_1 to dg_{19}). In particular, dg_5 contains the JPEG picture of the passport's holder. The standard specifies that JPEG pictures are of size 0 to 99999 bytes.

The Passive Authentication protocol works as follows. (1) The reader sends an authentication query, sending a pre-defined public value read, encrypted by the session key *ksenc* and MACed by the session MAC key *ksmac*. This ensures that the request comes from a legitimate reader. (2) The passport sends back the authentication information *dg* (from dg_1 to dg_{19}) together with a certificate $sod \stackrel{\mathsf{def}}{=} \mathsf{sign}(dg, sk_{DS})$, encrypted under the encryption key *ksenc* and MACed under *ksmac*. The certificate *sod* ensures the validity of the information.

5.2 Formal Specification of the Protocol

The formal specification of the Passive Authentication protocol is displayed in Figure 2. The process $PA(dg, \ell)$ represents a session of the passive authentication protocol, where *Pass* and *Reader* represent respectively the Passport Tag and the Reader. The key *ksenc* and *ksmac* are fresh names shared only by *Pass* and *Reader* since they are session keys previously established by the Basic Access Control protocol.

5.3 Unlinkability

The ICAO standard specifies that biometric passport must ensure *unlinkability*, *i.e.* must ensure that a user may make multiple uses of a service or a resource without others being able to link these uses together. The unlinkability of the Passive Authentication protocol can be formalised by the following equivalence:

$$\nu sk_{DS}.(PA(dg_1) \mid PA(dg_1) \approx^{\ell} \nu sk_{DS}.(PA(dg_1) \mid PA(dg_2))$$

$$
\begin{aligned}
Pass(dg, ksenc, ksmac) \stackrel{\text{def}}{=}\ & \mathsf{in}(c, x).\\
& \mathsf{if}\ \mathsf{mac}(\mathsf{proj}_1(x), ksmac) = \mathsf{proj}_2(x)\ \mathsf{then}\\
& \quad \mathsf{if}\ \mathsf{proj}_1(x) = \mathsf{senc}(\mathsf{read}, ksenc)\ \mathsf{then}\\
& \quad\quad \mathsf{let}\ y = \mathsf{senc}(\langle dg, \mathsf{sign}(dg, sk_{DS})\rangle, ksenc)\ \mathsf{in}\\
& \quad\quad \mathsf{out}(c, \langle y, \mathsf{mac}(y, ksmac)\rangle)\\
& \quad \mathsf{else}\ \mathsf{out}(c, Error)\\
& \mathsf{else}\ \mathsf{out}(c, Error)\\
Reader(ksenc, ksmac) \stackrel{\text{def}}{=}\ & \mathsf{let}\ xenc = \mathsf{senc}(\mathsf{read}, ksenc)\ \mathsf{in}\\
& \mathsf{out}(c, \langle xenc, \mathsf{mac}(xenc, ksmac)\rangle).\mathsf{in}(c, x).\\
& \mathsf{if}\ \mathsf{mac}(\mathsf{proj}_1(x), ksmac) = \mathsf{proj}_2(x)\ \mathsf{then}\\
& \quad \mathsf{let}\ y = \mathsf{sdec}(\mathsf{proj}_1(x), ksenc)\ \mathsf{in}\\
& \quad \mathsf{if}\ \mathsf{check}(\mathsf{proj}_2(y), \mathsf{vk}(sk_{DS})) = \mathsf{proj}_1(y)\ \mathsf{then}\ 0
\end{aligned}
$$

$$PA(dg) \stackrel{\text{def}}{=} \nu ksenc.\nu ksmac.(Pass(dg, ksenc, ksmac) \mid Reader(ksenc, ksmac))$$

Fig. 2. Formal specification of the Passive Authentication Protocol

where dg_1, dg_2 are the respective data groups of two passport. Intuitively, a user is unlinkable if an attacker cannot distinguish two sessions where the same user is present from two sessions where two different users are present.

Attack. Intuitively, the attack works as follows. We assume that the attacker first listens to an honest session between a reader and a passport A under attack. It therefore learns the size of the encryption of the data groups. Now, listening to any session between a reader and a passport B, it can compare the size of the encryption of the data groups. with the previous one. If they differ, A cannot be present, that is $B \neq A$. If they are equal, then B is likely to be A. How likely depends on the variability of the length and the size of the group of passport holders the attacker wish to distinguish from. Formally, this attack shows that $\nu sk_{DS}.(PA(dg_1) \mid PA(dg_1) \not\approx^{\ell} \nu sk_{DS}.(PA(dg_1) \mid PA(dg_2))$.

Impact. Our attack is very simple: a small device placed near a reader may very quickly decides whether A is present or not, simply listening to the messages received by the reader. [5] also describes an attack against unlinkability. It is based on the Basic Access Control protocol and relies on the fact that different error codes were used in the implementation of the French passports. The attack is dedicated to French passports and has now been fixed. Another attack demonstrated by A. Laurie consists in brute-forcing the document numbers of the passport (which normally requires to open and read the first page of the passport). Once the document numbers are known, anyone can access the data groups. In contrast, our attack does not require any access to these numbers and is inherent to the variability of the size of identifying objects such as pictures.

Fixes. The only simple fix is to ensure that data groups are of fixed size, typically by padding and/or restricting the range of size of data groups. However, this would result in heavier exchanges. Alternatively, a solution is to add padding of random size (which size varies at each transaction). The attacker would still gain some information on the probable user's identity but with smaller probability.

5.4 Implementation of the Decision Procedure

We have implemented our decision procedure in the active case (for the standard primitives) as an extension of the APTE tool [14]. Thanks to our tool, we can prove our fix (with padding) secure. Consider two data groups dg'_1, dg'_2 of the same length ($\ell(dg'_1) = \ell(dg'_2)$). Using APTE, we show that padding ensures unlinkability, that is, $\nu sk_{DS}.(PA(dg'_1) \mid PA(dg'_1) \approx^{\ell} \nu sk_{DS}.(PA(dg'_1) \mid PA(dg'_2))$. We can also show that our attack relies solely on the ability to compare lengths. Indeed, using APTE again, we can show that PA guarantees unlinkability for trace equivalence without length, that is $\nu sk_{DS}.(PA(dg_1) \mid PA(dg_1) \approx \nu sk_{DS}.(PA(dg_1) \mid PA(dg_2))$.

The following table summarises our findings using APTE on a 2.4 Ghz Intel Core 2 Duo. The input file used can be found in [14].

	Unlinkability	Time
PA w.r.t. $\approx$	true	4.42 sec
PA w.r.t. $\approx^{\ell}$	false	0.01 sec
PA with padding w.r.t. $\approx$	true	4.44 sec
PA with padding w.r.t. $\approx^{\ell}$	true	4.36 sec

6 Conclusion

We have proposed the first decision procedure for behavioral equivalence in presence of a length function. This allows e.g. to check for privacy properties more accurately. In the passive case, we have shown how to extend existing decidability results to a length function, for large classes of equational theories. In the active case, we provide a decision procedure for the standard primitives. Its implementation is an extension of the APTE tool [14]. As an application, we have discovered a new privacy flaw in the biometric passport. As future work, we plan to implement our attack and test it on several passports.

In this paper, we have focused on linear length functions since linear length functions can be realized for standard primitives and proved sound w.r.t. a cryptographic model [16]. We plan to investigate other families of length functions that are relevant for cryptographic primitives. In case some of these functions are not linear, we may need to revisit our procedure.

Protocols may sometimes perform length tests as well, for example, an agent may check that some data does not exceed a certain length. We believe that our procedure can be adapted in case length tests appear in the control flow of the protocols. It would require to extend the constraint systems used in the procedure in order to store constraints on the length. Adapting the decision procedure to solve these additional constraints might be challenging and raise difficult termination problems.

Our length function may also be used to capture other kind of leakages such as computation time or power consumption. To detect such side-channel attacks, we would need to model the "length" (or computation time / power consumption) of tests performed in the protocol. We plan to study whether our procedure can

be extended to the case where protocols not only leak the length of output terms but also the "length" of performed tests.

References

1. Machine readable travel document. Technical Report 9303, International Civil Aviation Organization (2008)
2. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. ACM Transactions on Information and System Security (TISSEC) 10(3), 1–59 (2007)
3. Abadi, M., Cortier, V.: Deciding knowledge in security protocols under equational theories. Theoretical Computer Science 387(1-2), 2–32 (2006)
4. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: 28th ACM Symp. on Principles of Programming Languages (POPL 2001) (2001)
5. Arapinis, M., Chothia, T., Ritter, E., Ryan, M.: Analysing unlinkability and anonymity using the applied pi calculus. In: 23rd IEEE Computer Security Foundations Symposium (CSF 2010) (2010)
6. Armando, A., et al.: The AVISPA Tool for the automated validation of internet security protocols and applications. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 281–285. Springer, Heidelberg (2005)
7. Bansal, C., Bhargavan, K., Maffeis, S.: Discovering concrete attacks on website authorization by formal analysis. In: 25th IEEE Computer Security Foundations Symposium (CSF 2012) (2012)
8. Baudet, M.: Deciding security of protocols against off-line guessing attacks. In: 12th Conference on Computer and Communications Security (CCS 2005) (2005)
9. Berrima, M., Ben Rajeb, N., Cortier, V.: Deciding knowledge in security protocols under some e-voting theories. Theoretical Informatics and Applications (RAIRO-ITA) 45, 269–299 (2011)
10. Blanchet, B.: An Efficient Cryptographic Protocol Verifier Based on Prolog Rules. In: 14th Computer Security Foundations Workshop (CSFW 2001) (2001)
11. Blanchet, B., Abadi, M., Fournet, C.: Automated verification of selected equivalences for security protocols. Journal of Logic and Algebraic Programming 75(1), 3–51 (2008)
12. Chadha, R., Ciobâcă, Ş., Kremer, S.: Automated verification of equivalence properties of cryptographic protocols. In: 21th European Symposium on Programming (ESOP 2012) (2012)
13. Cheval, V.: Automatic verification of cryptographic protocols: privacy-type properties. Phd thesis, ENS Cachan, France (2012)
14. Cheval, V.: APTE (Algorithm for Proving Trace Equivalence) (2013), `http://projects.lsv.ens-cachan.fr/APTE/`
15. Cheval, V., Comon-Lundh, H., Delaune, S.: Trace equivalence decision: Negative tests and non-determinism. In: 18th ACM Conference on Computer and Communications Security (CCS 2011) (2011)
16. Comon-Lundh, H., Cortier, V.: Computational soundness of observational equivalence. In: 15th Conf. on Computer and Communications Security (CCS 2008) (2008)
17. Cortier, V., Delaune, S.: A method for proving observational equivalence. In: 22nd IEEE Computer Security Foundations Symposium (CSF 2009) (2009)

Finding Security Vulnerabilities in a Network Protocol Using Parameterized Systems

Adi Sosnovich[1], Orna Grumberg[1], and Gabi Nakibly[2]

[1] Computer Science Department, Technion, Haifa, Israel
{sadisos,orna}@cs.technion.ac.il
[2] National EW Research and Simulation Center, Rafael, Haifa, Israel
gabin@rafael.co.il

Abstract. This paper presents a novel approach to *automatically* finding security vulnerabilities in the routing protocol OSPF – the most widely used protocol for Internet routing. We start by modeling OSPF on (concrete) networks with a fixed number of routers in a specific topology. By using the model checking tool CBMC, we found several simple, previously unpublished attacks on OSPF.

In order to search for attacks in a *family of networks* with varied sizes and topologies, we define the concept of an *abstract network* which represents such a family. The abstract network $\mathcal{A}$ has the property that if there is an attack on $\mathcal{A}$ then there is a corresponding attack on each of the (concrete) networks represented by $\mathcal{A}$.

The attacks we have found on abstract networks reveal security vulnerabilities in the OSPF protocol, which can harm routing in huge networks with complex topologies. Finding such attacks directly on the huge networks is practically impossible. Abstraction is therefore essential. Further, abstraction enables showing that the attacks are *general*. That is, they are applicable in a large (even infinite) number of networks. This indicates that the attacks exploit *fundamental vulnerabilities*, which are applicable to many configurations of the network.

1 Introduction

This paper presents a novel approach to automatically finding security vulnerabilities in the routing protocol *Open Shortest Path First* (OSPF) [14]. OSPF is the most widely used protocol for Internet routing, thus finding vulnerabilities which are inherent to the design of the protocol is significant for Internet security. Manually identifying vulnerabilities in a complex protocol such as OSPF is a hard task which requires deep understanding and close acquaintance with the protocol.

We propose to find vulnerabilities *automatically* by using model checking techniques. In order to use model checking for our purpose we build a model for the protocol when running on a given network topology; we include in the model an attacker with predefined capabilities; and we specify the absence of a state in which an attack succeeds (to be defined later). If the model checker finds a state violating the specification, it returns a counterexample leading to that state. The counterexample being a run of the protocol is, in fact, an *attack* on the protocol.

A high level description of the OSPF protocol is given below. OSPF runs on each router in a network of routers. Its goal is to distribute the full network topology to all

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 724–739, 2013.

routers. The routers send each other messages describing their partial view of the network topology. When a router gets a message from its neighbor, it updates its database accordingly and *floods* the message on to all of its *other* neighbors. OSPF includes a mechanism for fighting against possible attacks. If a router gets a message in its own name that it did not originate, then the router initiates a *"fight back"* message in order to correct the topology view of all other routers.

We start by modeling (concrete) networks with a fixed number of routers in a specific topology, where each router runs the OSPF protocol. The *attacker* is one of the routers running the same protocol, except that it can also send *fake* messages in the name of other routers, and can ignore messages sent to it. A *state* of the model consists of the databases and message queues of all routers in the network. We say that an *attack succeeds* in a state if (at least) one of the routers has a fake message in its database, and no router has a message waiting to be sent. This means that no fight back is going to change the fake topology view of this router. Thus, the attack is persistent.

We ran the model checking tool CBMC [2] on several topologies. We note that the OSPF protocol is quite elaborate. Further, the size of the database of each router is proportional to the size of the network. We therefore limited the topology sizes in order to fit in the model checker capacity. Nevertheless, we have found several simple, previously unpublished attacks. We also found a more subtle attack which was already published. The vulnerabilities revealed by the attacks we found are known and accepted by OSPF experts.

The limitation of the approach described so far is clear. It can only check a specific and small network topology which may expose only a part of the protocol's functionality. In order to allow for a good coverage of the protocol's functionality many other specific topologies need to be checked, taking more time and computing resources.

We therefore develop an approach which can search for attacks in a *parameterized network*, consisting of a *family of networks* with varied sizes and topologies. We define an *abstract network*, that represents such a family. The abstract network $\mathcal{A}$ has the property that if there is an attack on $\mathcal{A}$ then there is a corresponding attack on *each* of the (concrete) networks represented by $\mathcal{A}$. An abstract network allows to reveal security vulnerabilities in the OSPF protocol, which can harm routing in huge networks with complex topologies. Finding such attacks directly on the huge networks is practically impossible. Abstraction is therefore essential.

The abstraction is defined on all levels of the model: We define an abstract topology which represents a family of concrete topologies. An abstract state represents a set of concrete states. The correspondence between abstract transitions and their concrete counterpart is more subtle. Each abstract transition represents a set of finite concrete *runs*, one in each of the concrete topologies represented by A. As a result, our abstract model is unusual: It under-approximates each member in a family of concrete models. That is, every run of the abstract model has a corresponding run in each of the concrete models represented by it. This is an important characteristics of our abstraction as it allows us to find *general* attacks on an abstract network which are manifested in each of the concrete models it represents. Thus, these attacks are applicable in a large (even infinite) number of networks. This indicates that they exploit fundamental vulnerabilities, which are applicable to many configurations of the network. This is in contrast to

finding a specific attack that is only applicable for a single perhaps marginal network configuration.

In this part, we have found attacks on abstract networks manually. However, our abstract model can be implemented for instance in C to be used with CBMC, similarly to our implementation of the concrete model.

It should be noted that in principle, more attacks could be found on a concrete system that belongs to a family. However, in this work we are interested in finding general attacks, that are robust to changes in the topology. These are usually the first attacks a network operator would like to know with regard to its network.

We emphasize that the contributions of this work go beyond the security analysis of OSPF. The abstract concept and definition can be beneficial for finding security vulnerabilities in other protocols as well.
To summarize, the contributions of this work are:

- We analyzed the OSPF routing protocol and *automatically* found attacks on it.
- We found *general* attacks which are applicable to families of networks and demonstrated *security vulnerabilities* in the OSPF protocol.
- We developed a novel technique for *parameterized networks* which is suitable for finding a counterexample (in our case an attack) on each member of the family.
- This work is a first step towards finding security vulnerabilities in other distributed network protocols.

1.1 Related Work

There are a few works that present a security analysis of the OSPF protocol. Most such works (e.g., [17,18,7,15]) focus on LSA falsification attacks. Only two past works ([7] and [15]) present OSPF attacks with a persistent effect while evading a fight-back. This low number of works stands in contrast to the centrality of OSPF to Internet routing. This can be partially explained by the difficulty to do a manual and thorough security analysis of complex distributed network protocols.

There are some works that propose a security analysis of the design of network protocols based on model checking (e.g., [12,13,9]) . All past works check a given network configuration with a predetermined set of participants. In particular, some works (e.g., [11,5,10]) analyzed the security of OSPF and other routing protocols, while considering only a given network model. As other distributed network protocols the functionality of a routing protocol is highly dependent of the number of participants in the protocol and the network topology. Hence, current works that employ model checking for distributed network protocols may not cover the entire protocol's functionality.

Reasoning about families of systems, also known as *parameterized systems*, is a known research area (e.g. [6,8,4,16,1]). Most works present an abstract model which *over-approximates* all members in the family and is used to verify that they all satisfy a given property. We, on the other hand, define an abstract model which *under-approximates* each member in the family. Our abstract model is therefore most suitable for finding attacks on all members. To the best of our knowledge, no similar reasoning has been applied before to parameterized systems.

2 Modeling OSPF

2.1 OSPF Basics

The Internet is clustered into sets of connected networks and routers called Autonomous Systems (AS). Each AS is administered by a single authority, such as a large organization, or an Internet service provider. Within each AS a routing protocol is run. Its aim is to allow routers to construct their routing tables, while dynamically adapting to changes in the AS topology. Open Shortest Path First (OSPF) [14] is currently used within most ASes on the Internet. It was developed and standardized by the IETF organization.

Each OSPF router composes a list of all its links to neighboring routers and their costs. This list is termed *Link State Advertisement* (LSA). Each LSA is flooded throughout the AS. Every router compiles a database of the LSAs from all routers in the AS, thus having a complete view of the AS topology. This allows a router to calculate the least cost paths between it and every other router in the AS. As a result, the router's routing table is formed.

A new instance of each LSA is advertised periodically every 30 minutes, by default. Every LSA has a sequence number which is incremented with every new advertised instance. A more recent LSA instance with a higher sequence number will always take precedence over an older instance with a lower sequence number. An LSA includes the following fields: a) *src* - the router which just sent the LSA; b) *dest* - the router to which the LSA is destined; c) *orig* - the router which first advertised the LSA; d) *seq* - sequence number.

Two routers in the AS may be connected over a *point-to-point* link. A subset of two or more routers may be connected over a *transit network*. One router in every transit network is selected to act as a *designated router*. During the flooding of an LSA each router sends the LSA to all its neighbors (except the neighbor from which the LSA was received). To alleviate flooding load this rule has an exception: a non-designated router may flood an LSA over a transit network only to the designated router of that network. The designated router will send it to all the other routers in that transit network. Note that a router will only receive an LSA from one of its neighbors. An LSA having a *src* that is not one of the router's neighbors will be dropped.

A common goal for an OSPF attacker is to advertise a fake LSA on behalf of some other router in the AS. Such an attack changes the view other routers have of the AS topology and consequently changes their routing tables. The primary measure by which OSPF defends against such attacks is the "*fight-back*" mechanism. Once a victim router receives an instance of its own LSA which is newer than the last instance it originated, it immediately advertises a newer instance of the LSA with a higher sequence number which cancels out the false one. This mechanism prevents most OSPF attacks from persistently falsifying an LSA of another router. Another defense measure is the authentication of LSAs using a secret key shared by the routers of the autonomous system. An outside router that does not know the shared secret can not send LSAs to routers inside the autonomous system.

2.2 The Concrete Model

In the following we present the concrete model for OSPF we used to find attacks. We note that our model is a simplified version of the real OSPF.

Our model assumes as a starting point a stable routing state in the AS. Namely, all the routers advertised their LSAs and calculated their routing tables. In particular, no LSA flooding is in progress or about to start. The LSA databases of all routers are complete and identical. Without loss of generality we assume that the sequence numbers of all the LSAs that have been advertised are 0. In addition, designated routers for all transit networks have been selected. The model is composed of three entities: (AS) *topology* which models a concrete topology of the AS, *Router* which models a legitimate router inside the AS; *Attacker* which models a malicious router inside the AS.

Autonomous System Topology Model. We denote the concrete topology by $T_c = (R, S, E, DR_c)$, where R is the set of routers, $S \subseteq 2^R$ is the set of transit networks, which we refer to as *sub-network*, $E \subseteq R \times R$ is a set of undirected edges, each representing a point-to-point link between two routers, and $DR_c : S \to R$ maps sub-networks to their designated routers. For simplicity of presentation we assume that each router belongs to at least one sub-network. We emphasize that the routers forming a sub-network are directly connected to each other as if they were forming a clique. Nonetheless, those connections are not part of the set E which only includes point-to-point links. Figure 1 depicts an example of a topology.

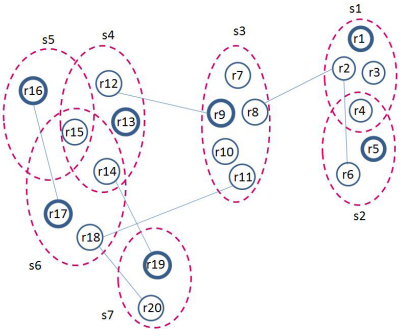


Fig. 1. The concrete topology T_C. The dashed circles marked as s_i are sub-networks, the circles r_i are routers, and lines connecting routers are edges. Bold circles represent designated routers.

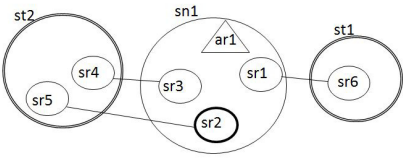


Fig. 2. Abstract topology T_A (see Section 3). The circles marked as sr_i represent singleton routers; the triangle ar_1 represents an abstract router; the circle sn_1 represents an abstract sub-network; and the double circles st_i represent sub-topologies. The bold circle represents a designated router (i.e., sr_2 is the designated router in the sub-network sn_1).

Router Model. The router model executes the standard functionality of the protocol. We model only part of the functionality defined by the OSPF standard since a large model might be infeasible for model checking. Nonetheless, our model captures the protocol's essential operations which any attack must exploit. For example, flooding by

its very nature must be exploited by any attack that aims to advertise false LSAs. The functionality we modeled includes: (1) LSA message structure. (2) Flooding procedure. (3) Designated router logic. (4) Fight-back mechanism.

We do not model the actual contents of each LSA, i.e. the list of advertised links and their costs, because the LSA content has no material affect on the attack technique used to advertise a fake LSA. Figure 3 gives a high level overview of the router procedure.

Attacker Model. In our work we assume that an attacker is one of the routers of the autonomous system. Other routers treat the attacker as a legitimate router. The attacker is free from the protocol's standard and is able to ignore incoming messages and to originate messages arbitrarily. In particular, an attacker may originate fake LSAs on behalf of other routers in the topology. The model indicates such LSAs by a special *isFake* flag, which is not part of the OSPF standard, and legitimate routers do not make use of it. This flag allows us to easily define the specifications for the model (see section 2.4). Note that since the attacker has control of a legitimate router, the attacker knows the secret key used to authenticate the LSA messages.

```
if (r.Q not empty)
{
  m = pop-head(r.Q)
  if (m.dest ≠ r)
    send m according to r's routing table
  else //m.dest is r
  {
    if (m is newer than the copy in r.DB)
    {
      if (m.orig == r)
        fight-back
      else
        update r.DB and flood m
    }
    else
      ignore m
  }
}
```

Fig. 3. A sketch of the router r procedure. $r.Q$ denotes r's incoming message queue. A message $m = (src, dest, orig, seq)$. $r.DB$ denotes the set of LSA instances currently installed in r's database.

Another important capability of the attacker is sending an LSA to a non-neighbor destination through several links without being opened on the way. Thus, the intermediate routers will not process the message. We call this *unicast* sending. This is a trivial capability that is inherent to any IP network. Every router (malicious or benign) can send messages directly to remote routers. However, regular routers following the OSPF protocol do not use this capability when flooding LSA messages.

2.3 Formal Model for OSPF

The formal model we use for OSPF is a finite state machine with global states and transitions. In order to obtain a finite model suitable for model checking, we impose a predefined bound SB on the sequence number of messages, and a predefined bound K on the queue size of each router. It should be noted that in real OSPF such bounds

exist as well. The queue of each router consists of up to K messages of the form $m = (src, dest, orig, seq, isFake)$, taken from the message domain $M = R \times R \times R \times \{0, ..., SB\} \times \{T, F\}$. The database of router r, $r.DB : R \to \{0, \ldots, SB\} \times \{T, F\}$, includes for each router r' the sequence number of the last message that was originated by r' and reached r , and the value of the flag *isFake* indicating whether this message was in fact originated by the attacker and not by r'. A global state $\sigma = \{r.DB \,|r \in R\} \cup \{r.Q \,|r \in R\}$ consists of a database and a message queue for each of the routers in the topology, including the attacker.

An *r-transition* between two global states corresponds to an application of the router r procedure (which is either the procedure given in Figure 3 if r is a regular router, or the attacker's procedure if r is the attacker). Note that an r-transition may change, in addition to the queue and the database of r, the queues of some of its neighbors. A *run* of the model consists of a sequence of global states $\sigma_1, \ldots \sigma_n$, such that for each i, a router r from R is chosen nondeterministically, and an r-transition is applied to σ_i, resulting in σ_{i+1}.

2.4 Specification

Our aim is to discover attacks on OSPF that allow an attacker to persistently falsify LSAs of legitimate routers. Our specification for the absence of a successful persistent attack requires that each state will satisfy at least one of the following two conditions:

1. No router has a fake LSA in its database.
2. At least one message resides in a router's queue.

The first condition verifies that the attacker has not fooled another router to install a fake LSA. The second condition relates to the attack's persistency. If not all the routers' queues are empty then the router whose LSA has been falsified might still fight back and revert the effect of the attack. Note that a state which violates the specification defines the outcome of a successful persistent attack regardless of a specific attack technique.

A model checker will search for a violation of the specification. When found, it will return a counterexample in the form of a run of the model which leads to a violating state. This run is actually an *attack* on OSPF.

2.5 Experimental Data

We have implemented in C our concrete model of OSPF, which is a simplified version of the protocol. The implementation is a rather small C program with a few hundreds of code lines. To find counterexamples, i.e. attacks, for which the above specification does not hold we use CBMC, a bounded model checker tool [2]. CBMC can check if a C program satisfies a specification along bounded

Table 1. For CNF formulas encoding topologies of different sizes, the number of variables and clauses in millions and the solving time in hours

#Routers	#Variables	#Clauses	Time
5	8M	21M	3.17h
6	17M	40M	7.07h
7	23M	55M	12.87h

runs. In our model, we bounded the number of cycles by 8, such that in each cycle any of the routers (including the attacker) can run their procedure once. In order to have a finite model which is rather small, we used a bound of $K = 4$ for the queue size, and a bound of $SB = 8$ for possible sequence numbers.

All our experiments were conducted on Intel Xeon X5650 with 32GB of memory. Table 1 details for several different network topologies of different sizes, the number of variables and clauses in the CNF formula generated by CBMC, and the time it took to solve the formula using the solver MiniSAT2 [3] .

2.6 Example of Attacks on OSPF

As mentioned before, when an attack is found the model checker CBMC outputs a path of global concrete states ending with a state that violates the specification. Figure 4 depicts an example of a topology with three sub-networks: $\{r1, r2\}$, $\{r3, r4\}$, and $\{r0\}$. $r1$ and $r4$ act as designated routers. The router $r0$ is attached to $r1$ and $r4$ using point-to-point links. In this topology $r3$ is the attacker. Note that although there are no edges between routers in the same sub-network, they are considered directly connected.

In the following we describe several attacks we found using the above concrete model having the topology depicted in Fig. 4. The first two attacks are simple albeit previously unpublished. The state explosion problem of the model checking impedes finding more complex attacks which may only be exhibited on larger topologies.

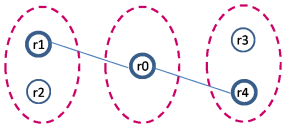


Fig. 4. A concrete topology

Recall that our model is a simplified version of the real OSPF. As the OSPF standard is given in an English manuscript, we cannot formally prove that our model is an under-approximation of the real OSPF. However, an OSPF expert validated that attacks found in our model are also valid in the full OSPF protocol.

Attack #1. The attacker ($r3$) originates a fake LSA on behalf of $r4$ directly to $r2$ (using unicast sending), while falsifying the source to be $r1$. The fields of the fake LSA are: $src = r1$, $dest = r2$, $orig = r4$, $seq = 1$, and *isFake* $= true$. $r2$ receives this LSA while considering it to be a valid LSA sent by $r1$. Since the sequence number of the attacker's LSA is larger than that of the LSA instance installed in $r2$'s database, $r2$ installs the attacker's LSA in its database. Since $r2$ received the message from $r1$, it does not flood it back to it. Since $r2$ has no other links no further messages are sent in the topology. Hence, the specification of our model is violated.

Attack #2. The following attack relies on the fact that the routers' queues are bounded. Note that any real-life router must bound its queue size that is dependent on the size of memory space in the router. The attacker continuously sends the following message many times: ($src = r3$, $dest = r4$, $orig = r0$, $seq = 1$, *isFake* $= true$). The number of sent copies should be larger than the bound on the size of the routers' queues.

The messages are received by $r4$ which floods the first message to $r0$. $r0$ then originates a fight-back message m' with $seq = 2$. Since the queue of $r4$ is full, m' will be discarded leaving $r4$ with the fake message installed in the database. All subsequent fake messages flooded to $r0$ will not trigger fight-back, since their sequence number (1) is smaller than that of the last message originated by $r0$ (m' with $seq = 2$). We note that the OSPF standard makes use of a reliable delivery of messages by leveraging acknowledgment messages. Hence a real router retransmits a message until it receives an acknowledgment. Our model does not include this functionality. Nonetheless, this attack would still be feasible in real life if the attacker continued sending messages to keep $r4$'s queue full.

Attack #3. The following attack was first described in [15]. The attacker sends the following two LSA messages: $m1$ = ($src = r3$, $dest = r4$, $orig = r1$, $seq = 1$, *isFake* $= true$) and $m2$ = ($src = r3$, $dest = r4$, $orig = r1$, $seq = 2$, *isFake* $= true$). First, $m1$ is received and installed by $r4$. Then, $r4$ floods it to $r0$. Afterward, $m2$ is received by $r4$. Since it has a higher sequence number than $m1$, $m2$ supersedes it in $r4$'s database. $m2$ is also flooded to $r0$. $r0$ processes and sends both messages to $r1$, while $m2$ is the last to be installed in its database. Once $r1$ receives $m1$ it immediately originates a fight-back message $m3$ with $seq = 2$ and floods it to all its neighbors. $r1$ then receives $m2$. Since $m2$ and $m3$ have equal sequence number (2), $m2$ is not considered newer than $m3$, hence $r1$ does not send another fight-back message and ignores $m2$. Once $r0$ receives $m3$, it does not consider it newer than $m2$ which is currently installed in its database. Hence, it ignores $m3$. Since $r4$ installed the fake message $m2$ and no more messages are waiting to be sent the specification of our model is violated.

3 An Abstract Network and Its Matching Concrete Networks

In the previous section we showed how attacks can be found on concrete models. Due to the state explosion problem, the models that can be handled are very small in size and hence restricted in their topologies. We would like to extend our search for attacks to larger and more complex topologies. Further, we are interested in *general* attacks, which are insensitive to most of the topology's details and therefore can be applied in a family of topologies.

In order to achieve that, we define an *abstract model* which can represent a family of concrete models. The models in the family are similar in some aspects of their topologies but may differ in many other aspects.

The abstract model consists of an abstract topology which includes abstract components representing a large number of routers and sub-networks, and of an abstract protocol which is an adjustment of OSPF to the abstract components.

We define several level of abstract components. The most abstract component is the *sub-topology*, which represents any number of concrete sub-networks. The edges between the sub-topology and the rest of the topology are not abstracted. As a result, routers within the sub-topology which are connected to these edges remain unabstracted as well. These routers are called *singleton routers*. The concrete routers they

represent are called *visible*. All other routers within the sub-topology and the edges among them are fully abstracted, and are referred to as *invisible*.

Another abstract component is the *abstract router* which represents a set of concrete routers, all contained within the same sub-network, and have no edges outside of the sub-network. An *abstract sub-network* consists of a set of abstract routers and a set of singleton routers. As with sub-topologies, the singleton routers in a sub-network are un-abstracted. They represent a single concrete router whose edges are un-abstracted too. We require that each singleton router belongs to either a sub-topology or a nonempty set of abstract sub-networks.

The intuition behind the definition of an abstract topology is as follows. The un-abstracted routers are those that may participate in an attack. The others are needed to form a topology that brings unabstracted routers to manifest more of their OSPF functionality and thus to possibly expose more security vulnerabilities. Moreover, abstracted routers allow to show that a found attack is general and applicable to a family of topologies.

Clearly, the attacker is always an (un-abstracted) singleton router. Moreover, the messages sent by the attacker are un-abstracted as well. That is, their originator, source, and destination fields refer to singleton routers.

We impose some constraints on abstract sub-topologies, to guarantee that for every abstract transition and every concrete topology represented by the abstract topology, there can be found a corresponding finite concrete run.

For a sub-topology st, recall that each singleton router in st represents a single concrete visible router. We require that in the part of the concrete topology which is represented by st, each of its visible routers must belong to a different sub-network. Also, visible routers in st may not be directly connected to each other, but should be connected to at least one invisible router. Further, the invisible routers in st form a strongly connected component. These constraints guarantee that if a message is flooded to st by a singleton router r, then there is a concrete run along which the message is opened by all invisible routers prior to being opened by any other singleton router.

While these constraints seem quite restrictive, our abstract topologies still represent a large variety of topologies of different sizes. As shown in Section 5, some nontrivial attacks were found on them. Many of these constraints can be removed for the price of much more complex definitions and correctness proof. We choose to present a simpler version here, and to demonstrate its usability.

3.1 Abstract Topology

Formally, an abstract topology is denoted by $T_A = (SR, ST, AR, SN, E_A, DR_A)$ where, SR is a set of abstract singleton routers, $ST \subseteq 2^{SR}$ is a set of sub-topologies, AR is a set of abstract routers, $SN \subseteq 2^{AR \cup SR}$ is a set of abstract sub-networks, and $E_A \subseteq SR \times SR$ is a set of undirected edges, each representing a point-to-point link between two abstract singleton routers. Finally, $DR_A : SN \rightarrow SR$ is a function that maps sub-networks to their designated router, which must be from SR. Figure 2 presents an abstract topology. Note that, similarly to the concrete case, connections between routers within the same sub-network are not depicted in the figure.

3.2 Matching Abstract and Concrete Topologies

Next we define a matching relation between abstract and concrete topologies. The matching relation adhere to the intuitive explanation given above. Let $T_A = (SR, ST, AR, SN, E_A, DR_A)$ be an abstract topology and $T_C = (R, S, E, DR_C)$ be a concrete topology. A relation

$$H \subseteq (SR \times R) \cup \left(AR \times 2^R\right) \cup (SN \times S) \cup \left(ST \times 2^S\right) \cup (E_A \times E)\,.$$

is a *matching relation* between T_A and T_C if it satisfies the following constraints:

- H restricted to each one of its domains is a 1-1 function. For instance, $H \cap (SR \times R)$ is a 1-1 function. By abuse of notation we refer to it as $H : SR \to R$.
- A sub-topology st represents a set of concrete sub-networks S'. Each singleton router in st is matched to a concrete router in a sub-network in S'. Different singleton routers in st are matched to routers in different sub-networks in S'.
- An abstract sub-network sn represents a concrete sub-network s such that each singleton router in sn is matched to a router in s, and each abstract router in sn is matched to a set of routers in s. Every router in s has a matched component in sn.
- Each concrete sub-network is matched to either an abstract sub-network or a sub-topology.
- There is an abstract edge between two singleton routers if and only if there is a concrete edge between their matched routers.

For example, the relation H, given below, is a matching relation between T_A from Figure 2 and T_C from Figure 1.

- $H \cap (SR \times R) = \{(sr1, r8)\,, (sr2, r9)\,, (sr3, r11)\,, (sr4, r18)\,, (sr5, r12)\,, (sr6, r2)\}$
- $H \cap \left(AR \times 2^R\right) = \{(ar1, \{r7, r10\})\}$
- $H \cap (SN \times S) = \{(sn1, s3)\}$
- $H \cap \left(ST \times 2^S\right) = \{(st1, \{s1, s2\})\,, (st2, \{s4, s5, s6, s7\})\}$
- $H \cap (E_A \times E) = \{((sr1, sr6)\,, (r8, r2))\,, ((sr4, sr3)\,, (r18, r11))\,, ((sr2, sr5)\,, (r9, r12))\}$

3.3 Global Abstract States

Let T_A be an abstract topology and let $AC = ST \cup AR \cup SR$ be the set of components in the abstract topology. The message domain in the abstract model is $M = AC \times AC \times ORIGS \times \{0, ..., SB\} \times \{T, F\}$, where $ORIGS \subseteq SR$ is a predefined set of originators which can be used by the attacker in its messages. Abstract messages consist of the same fields as concrete messages.

An abstract state is defined by $\sigma_A = \{ac.DB \,|ac \in AC\} \cup \{ac.Q \,|ac \in AR \cup SR\}$, where for every component $ac \in AC$, the structure of its database is identical to that of a concrete component, $ac.DB : ORIGS \to \{0, ..., SB\} \times \{T, F\}$, except that here it is only defined for the subset $ORIGS \subseteq SR$. In addition, for every $ac \in AR \cup SR$, $ac.Q$ is a queue of up to K messages. The database is restricted to $ORIGS$ since in

our setting (see section 2.2) only the attacker originates messages, and those messages have $orig \in ORIGS$. Thus, there is no need for $ac.DB$ to contain entries of other originators.

Note that, we do not define a queue for sub-topologies st, since flooding within st is always described as a single abstract transition. Each singleton router in st has a queue. Thus, a queue for st would have represented the queues of all invisible routers, matched to st. However, the queues of all invisible routers are empty whenever the abstract transition begins or ends. Thus, there is no need to represent their content.

3.4 Matching Abstract and Concrete States

Let T_A and T_C be an abstract and concrete topologies and let H be their matching relation. In order to define a matching between abstract and concrete states, we first define a matching between abstract and concrete databases and queues.

We use h to denote a function that matches abstract databases, messages, queues, and global states to sets of their concrete counterparts.

1. An abstract database DB_A matches a concrete database DB_C, denoted $DB_C \in h(DB_A)$, if for each $o \in ORIGS$, the entry for o in DB_A is identical to the entry of $H(o)$ in DB_C.
2. An abstract message m and a concrete message m' match, denoted $m' \in h(m)$, if $m'.src \in H(m.src)$, $m'.dest \in H(m.dest)$, $m'.orig = H(m.orig)$, $m'.seq = m.seq$, and $m'.isFake = m.isFake$.
   Since $orig$ is a singleton router and since seq and $IsFake$ are un-abstracted, they have a single matching.
3. An abstract queue matches a concrete queue if
   (a) For a singleton router sr, each message m in its queue is matched with a sequence of (one or more) concrete messages in $h(m)$.
   The reason for matching more than one concrete message with m is that an abstract transition may add only one message to the queue. On the other hand, the concrete run that correspond to this transition consists of several concrete transitions, each of which may add a matching message to the queue. This is because, when sr is part of a sub-topology st, then the invisible routers represented by st may flood the message several times to sr, via different paths in the sub-topology.
   (b) For an abstract router ar, its queue represents the queues of all concrete routers matched with ar. Here the sizes of the queues are identical since a message received by ar corresponds to single messages received by each r in $H(ar)$ from the designated router. No other messages are sent among routers in $H(ar)$.

We can now define matching of abstract and concrete states. $\sigma_C \in h(\sigma_A)$ if the following conditions holds

1. $\forall ac \in AR \cup SR\,[\forall r \in H(ac)\,(r.Q \in h(ac.Q))]$. That is, queues of matching components must match.
2. $\forall ac \in SR \cup ST \cup AR\,[\forall r \in H(ac)\,(r.DB \in h(ac.DB))]$. That is, databases of matching components must match.

3.5 Abstract Transitions and Their Matching Concrete Transitions

Similarly to the concrete model, an *abstract transition* between two global abstract states corresponds to an application of the procedure of one of the abstract components. The abstract model includes procedures for a singleton router, an abstract router, and an attacker. Our model does not include a procedure for a sub-topology. Instead, its behavior is defined as part of the procedure of singleton routers included in it.

A high-level description of the procedure of a singleton router sr is given in Figure 5. It is similar to the procedure of a concrete router, except that it does not handle messages whose destination is not sr. This is because in the abstract model such messages are sent by unicast directly to their destination. The singleton router procedure can perform either flooding or fight back. Figure 6 describes the flooding procedure performed by a singleton router (as part of its procedure). $FD_A(sr, m.src)$ returns the flooding destinations, i.e. set of abstract components to which sr floods a message m obtained from component src. The fight back procedure is similar, except that FD_A is replaced by the fight back destinations, FBD_A. The statement $ac_1.Q' = ac_1.Q \bullet \{m_{sr \to ac_1}\}$ performs an update of ac_1's queue. The resulting queue, $ac_1.Q'$, is obtained by concatenating the old queue $ac_1.Q$ with a message which is identical to m, except that its src is sr and its destination is ac_1.

The procedure of an abstract router is simpler. It only installs a message from its queue in its database and does not perform flood or fight back. This is because it is part of a single abstract sub-network, and is not connected by any edges.

An ac-abstract transition corresponds to a single application of the procedure for abstract component ac. This transition may represent either a single concrete transition or a sequence of concrete transitions (i.e., a concrete run), depending on the type of ac and on the message content. Below we detail a few non-trivial cases where abstract transitions correspond to a concrete run. For every concrete topology T_C represented by an abstract topology T_A and for every abstract transition in T_A, a corresponding concrete run as detailed below can be found in T_C.

Case 1. Consider an abstract transition in which a singleton router sr floods a message m, where sr is within a sub-topology st, and st belongs to the flooding destinations of sr. In such a case, the concrete run represented by the abstract transition includes, in addition to the flooding done by sr, the flooding applied by the invisible routers in $H(st)$. By the end of this run, all invisible routers within st have already removed m from their queue, updated their databases (if their databases were less updated), and flooded m further to the rest of the visible routers in $H(st)$.

Case 2. Consider an abstract transition in which a singleton router sr in a sub-topology st floods a message m, where $m.src = st$. This abstract transition represents a concrete run in which $H(sr)$ floods m. In addition, invisible routers in $H(st)$, which are included in the flooding destinations of $H(sr)$, remove m from their queue and ignore it.

Case 3. Consider an abstract transition in which the attacker sends a message m by unicast to a destination which is not one of its neighbors. That is, the message m is added

to the queue of its destination. This abstract transition represents a sequence of concrete transitions in which each router on the routing path which is not the destination, sends the message according to its routing table, without opening the message.

Case 4. Abstract transition taken by an abstract router ar represents a sequence of similar concrete transitions taken by each of the concrete routers represented by ar exactly once.

```
singleton router procedure(sr)
if (sr.Q not empty)
{
  m = pop-head(Q)
    if (m is newer than the copy in sr.DB)
    {
      if (m.orig == sr)
        fight back(sr, m)
      else
        update sr.DB and flood(sr, m)
    }
    else
      ignore m
  }
}
```

Fig. 5. Procedure of a singleton router

```
flood(sr, m)
For each
ac_1 ∈ FD_A (sr, m.src) ∩ (AR ∪ SR))
{
  ac_1.Q' = ac_1.Q • {m_{sr→ac_1}}
}
For each st ∈ FD_A (sr, m.src) ∩ ST
{
  if (st.DB[m.orig].seq < m.seq)
  {
    st.DB'[m.orig] = (m.seq, m.isFake)
    For each sr_1 ∈ FD_A (st, sr)
      sr_1.Q' = sr_1.Q • {m_{st→sr_1}}
  }
}
```

Fig. 6. flooding procedure of a singleton router sr, where m is the message to flood

4 Correctness of the Algorithm

Theorem 1. *Let T_A and T_C be an abstract and concrete topologies and let H be their matching relation. Then, for each finite abstract run $\sigma_1, \ldots \sigma_n$, there exists a corresponding finite concrete run $\sigma'_1, \ldots \sigma'_k$, such that $\sigma'_1 \in h(\sigma_1)$ and $\sigma'_k \in h(\sigma_n)$.*

Corollary 1. *An abstract attack found on an abstract topology T_A, has a corresponding attack on each matching topology T_C .*

Proof Sketch

- We show that for each abstract transition, there is a concrete finite run, such that the initial and final states of the transition and of the run are matching.
- An abstract attack is an abstract run for which the final state violates our specification. A concrete state matching an abstract state which violates the specification, also violates the specification. Thus, the corresponding paths are concrete attacks.
- The proof is based on the matching relation H and on the function h, defined in section 3.

5 Examples of Attacks on OSPF in the Abstract Model

In this section we describe a few attacks, found on different abstract models which we picked manually.

Attack #1. This attack has been found on the abstract topology T_A, presented in Figure 2. The attacker is $sr2$. The set of predefined originators is $ORIGS = \{sr1\}$. The attacker originates a fake message on behalf of $sr1$: $m = (src = sr2, dest = sr5, orig = sr1, seq = 1, isFake = T)$. $sr5$ receives this message while considering it to be a valid message, sent by $sr2$. Since the sequence number of m is larger than that of the message instance installed in $sr5$'s database, $sr5$ installs m in its database, and floods it. The fake message will be flooded and installed in the databases of $st2, sr4$, and $sr3$. When m is installed by sr3, it will be flooded to the attacker $sr2$, since $sr2$ is the designated router of the sub-network $sn1$. The attacker will choose to ignore m, thus preventing this message from being flooded to $sr1$, and avoiding fight back. Since no more messages are waiting to be sent, the specification is violated.

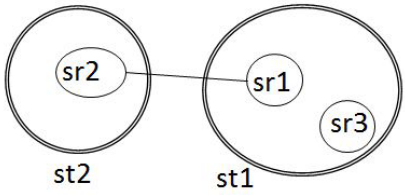


Fig. 7. Abstract topology on which attack #2 is described

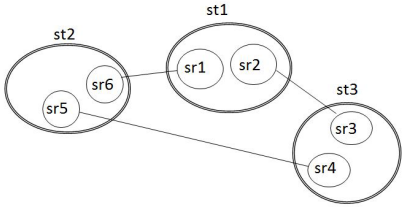


Fig. 8. Abstract topology on which attack #3 is described

Attack #2. T_A is the abstract topology presented in Figure 7. The attacker is $sr3$. The set of predefined originators is $ORIGS = \{sr1\}$. The attacker originates a fake message on behalf of $sr1$: $m = (src = sr1, dest = sr2, orig = sr1, seq = 1, isFake = T)$, which is sent by unicast to $sr2$. $sr2$ installs the fake message in its database and floods it only to the sub-topology $st2$ due to the flooding rules of OSPF. Therefore, in the final state the queues of all abstract components are empty, and the databases of $sr2$ and $st2$ are installed with the fake message. Thus, the specification is violated.

Attack #3. T_A is the abstract topology presented in Figure 8. The attacker is $sr3$. The set of predefined originators is $ORIGS = \{sr2\}$. The attacker sends the following two LSAs (using unicast sending): $m1 = (src = sr3, dest = sr2, orig = sr2, seq = 1, isFake = T)$ and $m2 = (src = sr4, dest = sr5, orig = sr2, seq = 2, isFake = T)$. As a result, $sr2$ sends a fight back message $m3$ with $orig = sr2, seq = 2, isFake = F$, but $sr5$ opens $m3$ after it has already installed $m2$ in its database, and will thus ignore the fight back message and will remain with the fake message.

6 Directions for Future Research

An important direction for future research is to generalize the method for finding general attacks applicable to families of network topologies to other network protocols, in particular routing protocols. Another direction is to develop a methodology for deciding which abstract networks to check, and to automate the abstraction process. Additional direction is to extend the abstraction mechanism for finding attacks which are applicable to a sub-family rather than the whole family, to enable finding more possible attacks.

Acknowledgement. We Thank Manfred Grumberg for initiating the project. This research was conducted as part of the KABARNIT consortium, with the support of the MAGNET founds.

References

1. Abdulla, P.: Regular model checking. STTT 14(2) (2012)
2. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
3. Niklas Een, N.S.: Minsat 2.0 - (2008), `http://minisat.se/minisat.html`
4. Emerson, E.A., Kahlon, V.: Exact and efficient verification of parameterized cache coherence protocols. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 247–262. Springer, Heidelberg (2003)
5. Fortz, B.: On the evaluation of the reliability of OSPF routing in IP networks. Technical report, Institut dadministration et de gestion (2001)
6. German, S., Sistla, P.: Reasoning about systems with many processes. J. ACM 39(3) (1992)
7. Jones, E., Le Moigne, O.: OSPF security vulnerabilities analysis. Internet-Draft draft-ietf-rpsec-ospf-vuln-02, IETF (June 2006)
8. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich ssertional languages. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997)
9. Liu, J., Ye, X., Zhang, J., Li, J.: Security verification of 802.11i 4-way handshake protocol. In: Communications (2008)
10. Malik, S.U.R., Srinivasan, S.K., Khan, S.U., Wang, L.: A methodology for OSPF routing protocol verification. In: 12th International Conference on Scalable Computing and Communications (ScalCom) (2012)
11. Matousek, P., Ráb, J., Rysavy, O., Svéda, M.: A formal model for network-wide security analysis. In: Engineering of Computer Based Systems (2008)
12. John, C.: Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murphi. In: IEEE Symposium on Security and Privacy, pp. 141–151 (1997)
13. Mitchell, J.C., Roy, A., Rowe, P., Scedrov, A.: Analysis of EAP-GPSK Authentication Protocol. In: Bellovin, S.M., Gennaro, R., Keromytis, A.D., Yung, M. (eds.) ACNS 2008. LNCS, vol. 5037, pp. 309–327. Springer, Heidelberg (2008)
14. Moy, J.: OSPF version 2. IETF RFC 2328 (April 1998)
15. Nakibly, G., Gonikman, D., Kirshon, A., Boneh, D.: Persistent OSPF attacks. In: NDSS (2012)
16. Saksena, M., Wibling, O., Jonsson, B.: Graph grammar modeling and verification of ad hoc routing protocols. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 18–32. Springer, Heidelberg (2008)
17. Wang, F., Vetter, B., Wu, S.F.: Secure routing protocols: Theory and practice. Technical report, North Carolina State University (May 1997)
18. Wu, S.F., et al.: JiNao: Design and implementation of a scalable intrusion detection system for the OSPF routing protocol. ACM Transactions on Computer Systems 2 (1999)

Fully Automated Shape Analysis Based on Forest Automata

Lukáš Holík, Ondřej Lengál, Adam Rogalewicz, Jiří Šimáček, and Tomáš Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic

Abstract. Forest automata (FA) have recently been proposed as a tool for shape analysis of complex heap structures. FA encode sets of tree decompositions of heap graphs in the form of tuples of tree automata. In order to allow for representing complex heap graphs, the notion of FA allowed one to provide user-defined FA (called boxes) that encode repetitive graph patterns of shape graphs to be used as alphabet symbols of other, higher-level FA. In this paper, we propose a novel technique of automatically learning the FA to be used as boxes that avoids the need of providing them manually. Further, we propose a significant improvement of the automata abstraction used in the analysis. The result is an efficient, fully-automated analysis that can handle even as complex data structures as skip lists, with the performance comparable to state-of-the-art fully-automated tools based on separation logic, which, however, specialise in dealing with linked lists only.

1 Introduction

Dealing with programs that use complex dynamic linked data structures belongs to the most challenging tasks in formal program analysis. The reason is a necessity of coping with infinite sets of reachable heap configurations that have a form of complex graphs. Representing and manipulating such sets in a sufficiently general, efficient, and automated way is a notoriously difficult problem.

In [6], a notion of *forest automata* (FA) has been proposed for representing sets of reachable configurations of programs with complex dynamic linked data structures. FA have a form of tuples of *tree automata* (TA) that encode sets of heap graphs decomposed into tuples of *tree components* whose leaves may refer back to the roots of the components. In order to allow for dealing with complex heap graphs, FA may be *hierarchically nested* by using them as alphabet symbols of other, higher-level FA. Alongside the notion of FA, a shape analysis applying FA in the framework of *abstract regular tree model checking* (ARTMC) [2] has been proposed in [6] and implemented in the Forester tool. ARTMC accelerates the computation of sets of reachable program configurations represented by FA by abstracting their component TA, which is done by collapsing some of their states. The analysis was experimentally shown to be capable of proving memory safety of quite rich classes of heap structures as well as to be quite efficient. However, it relied on the user to provide the needed nested FA—called *boxes*—to be used as alphabet symbols of the top-level FA.

In this paper, we propose a new shape analysis based on FA that avoids the need of manually providing the appropriate boxes. For that purpose, we propose a technique of automatically *learning* the FA to be used as boxes. The basic principle of the learning

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 740–755, 2013.

stems from the reason for which boxes were originally introduced into FA. In particular, FA must have a separate component TA for each node (called a *join*) of the represented graphs that has multiple incoming edges. If the number of joins is unbounded (as, e.g., in doubly linked lists, abbreviated as DLLs below), unboundedly many component TA are needed in flat FA. However, when some of the edges are hidden in a box (as, e.g., the prev and next links of DLLs in Fig. 1) and replaced by a single box-labelled edge, a finite number of component TA may suffice. Hence, the basic idea of our learning is to identify subgraphs of the FA-represented graphs that contain at least one join, and when they are enclosed—or, as we say later on, *folded*—into a box, the in-degree of the join decreases.

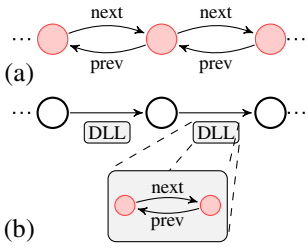


Fig. 1. (a) A DLL, (b) a hierarchical encoding of a DLL

There are, of course, many ways to select the above mentioned subgraphs to be used as boxes. To choose among them, we propose several criteria that we found useful in a number of experiments. Most importantly, the boxes must be *reusable* in order to allow eliminating as many joins as possible. The general strategy here is to choose boxes that are *simple* and *small* since these are more likely to correspond to graph patterns that appear repeatedly in typical data structures. For instance, in the already mentioned case of DLLs, it is enough to use a box enclosing a single pair of next/prev links. On the other hand, as also discussed below, too simple boxes are sometimes not useful either.

Further, we propose a way how box learning can be efficiently integrated into the main analysis loop. In particular, we do not use the perhaps obvious approach of incrementally building a *database of boxes* whose instances would be sought in the generated FA. We found this approach inefficient due to the costly operation of finding instances of different boxes in FA-represented graphs. Instead, we always try to identify which subgraphs of the graphs represented by a given FA could be folded into a box, followed by looking into the so-far built database of boxes whether such a box has already been introduced or not. Moreover, this approach has the advantage that it allows one to use simple language inclusion checks for *approximate box folding*, replacing a set of subgraphs that appear in the graphs represented by a given FA by a larger set, which sometimes greatly accelerates the computation. Finally, to further improve the efficiency, we interleave the process of box learning with the *automata abstraction* into a single iterative process. In addition, we propose an FA-specific improvement of the basic automata abstraction which *accelerates the abstraction* of an FA using components of other FA. Intuitively, it lets the abstraction synthesize an invariant faster by allowing it to combine information coming from different branches of the symbolic computation.

We have prototyped the proposed techniques in Forester and evaluated it on a number of challenging case studies. The results show that the obtained approach is both quite general as well as efficient. We were, e.g., able to fully-automatically analyse programs with 2-level and 3-level skip lists, which, according to the best of our knowledge, no other fully-automated analyser can handle. On the other hand, our implementation achieves performance comparable and sometimes even better than that of Predator [4]

(a winner of the heap manipulation division of SV-COMP'13) on list manipulating programs despite being able to handle much more general classes of heap graphs.

Related Work. As discussed already above, we propose a new shape analysis based upon the notion of forest automata introduced in [6]. The new analysis is extended by a mechanism for automatically learning the needed nested FA, which is carefully integrated into the main analysis loop in order to maximize its efficiency. Moreover, we formalize the abstraction used in [6], which was not done in [6], and subsequently significantly refine it in order to improve both its generality as well as efficiency.

From the point of view of efficiency and degree of automation, the main alternative to our approach is the fully-automated use of separation logic with inductive list predicates as implemented in Space Invader [12] or SLAyer [1]. These approaches are, however, much less general than our approach since they are restricted to programs over certain classes of linked lists (and cannot handle even structures such as linked lists with data pointers pointing either inside the list nodes or optionally outside of them, which we can easily handle as discussed later on). A similar comparison applies to the Predator tool inspired by separation logic but using purely graph-based algorithms [4]. The work [9] on overlaid data structures mentions an extension of Space Invader to trees, but this extension is of a limited generality and requires some manual help.

In [5], an approach for synthesising inductive predicates in separation logic is proposed. This approach is shown to handle even tree-like structures with additional pointers. One of these structures, namely, the so-called mcf trees implementing trees whose nodes have an arbitrary number of successors linked in a DLL, is even more general than what can in principle be described by hierarchically nested FA (to describe mcf trees, recursively nested FA or FA based on hedge automata would be needed). On the other hand, the approach of [5] seems quite dependent on exploiting the fact that the encountered data structures are built in a "nice" way conforming to the structure of the predicate to be learnt (meaning, e.g., that lists are built by adding elements at the end only), which is close to providing an inductive definition of the data structure.

The work [10] proposes an approach which uses separation logic for generating numerical abstractions of heap manipulating programs allowing for checking both their safety as well as termination. The described experiments include even verification of programs with 2-level skip lists. However, the work still expects the user to manually provide an inductive definition of skip lists in advance. Likewise, the work [3] based on the so-called separating shape graphs reports on verification of programs with 2-level skip lists, but it also requires the user to come up with summary edges to be used for summarizing skip list segments, hence basically with an inductive definition of skip lists. Compared to [10,3], we did not have to provide any manual aid whatsoever to our technique when dealing with 2-level as well as 3-level skip lists in our experiments.

A concept of inferring graph grammar rules for the heap abstraction proposed in [8] has recently appeared in [11]. However, the proposed technique can so far only handle much less general structures than in our case.

2 Forest Automata

Given a word $\alpha = a_1 \dots a_n, n \geq 1$, we write α_i to denote its i-th symbol a_i. Given a total map $f : A \to B$, we use $dom(f)$ to denote its domain A and $img(f)$ to denote its image.

Graphs. A *ranked alphabet* is a finite set of symbols Σ associated with a mapping $\# : \Sigma \to \mathbb{N}_0$ that assigns ranks to symbols. A (directed, ordered, labelled) *graph* over Σ is a total map $g : V \to \Sigma \times V^*$ which assigns to every *node* $v \in V$ (1) a *label* from Σ, denoted as $\ell_g(v)$, and (2) a sequence of *successors* from V^*, denoted as $S_g(v)$, such that $\#\ell_g(v) = |S_g(v)|$. We drop the subscript g if no confusion may arise. Nodes v with $S(v) = \varepsilon$ are called *leaves*. For any $v \in V$ such that $g(v) = (a, v_1 \cdots v_n)$, we call the pair $v \mapsto (a, v_1 \cdots v_n)$ an *edge* of g. The *in-degree* of a node in V is the overall number of its occurrences in $g(v)$ across all $v \in V$. The nodes of a graph g with an in-degree larger than one are called *joins* of g.

A *path* from v to v' in g is a sequence $p = v_0, i_1, v_1, \ldots, i_n, v_n$ where $v_0 = v$, $v_n = v'$, and for each $j : 1 \leq j \leq n$, v_j is the i_j-th successor of v_{j-1}. The *length* of p is defined as $\mathit{length}(p) = n$. The *cost* of p is the sequence $i_1, \ldots, i_n$. We say that p is cheaper than another path p' iff the cost of p is lexicographically smaller than that of p'. A node u is *reachable* from a node v iff there is a path from v to u or $u = v$. A graph g is *accessible* from a node v iff all its nodes are reachable from v. The node v is then called the *root* of g. A *tree* is a graph t which is either empty, or it has exactly one root and each of its nodes is the i-th successor of at most one node v for some $i \in \mathbb{N}$.

Forests. Let $\Sigma \cap \mathbb{N} = \emptyset$. A Σ-labelled *forest* is a sequence of trees $t_1 \cdots t_n$ over $(\Sigma \cup \{1, \ldots, n\})$ where $\forall 1 \leq i \leq n : \#i = 0$. Leaves labelled by $i \in \mathbb{N}$ are called *root references*.

The forest $t_1 \cdots t_n$ represents the graph $\otimes t_1 \cdots t_n$ obtained by uniting the trees of $t_1 \cdots t_n$, assuming w.l.o.g. that their sets of nodes are disjoint, and interconnecting their roots with the corresponding root references. Formally, $\otimes t_1 \cdots t_n$ contains an edge $v \mapsto (a, v_1 \cdots v_m)$ iff there is an edge $v \mapsto (a, v'_1 \cdots v'_m)$ of some tree $t_i, 1 \leq i \leq n$, s.t. for all $1 \leq j \leq m$, $v_j = \mathit{root}(t_k)$ if v'_j is a root reference with $\ell(v'_j) = k$, and $v_j = v'_j$ otherwise.

Tree Automata. A (finite, non-deterministic, top-down) *tree automaton* (TA) is a quadruple $A = (Q, \Sigma, \Delta, R)$ where Q is a finite set of *states*, $R \subseteq Q$ is a set of *root states*, Σ is a ranked alphabet, and Δ is a set of *transition rules*. Each transition rule is a triple of the form $(q, a, q_1 \ldots q_n)$ where $n \geq 0$, $q, q_1, \ldots, q_n \in Q$, $a \in \Sigma$, and $\#a = n$. In the special case where $n = 0$, we speak about the so-called *leaf rules*.

A *run* of A over a tree t over Σ is a mapping $\rho : \mathit{dom}(t) \to Q$ s.t. for each node $v \in \mathit{dom}(t)$ where $q = \rho(v)$, if $q_i = \rho(S(v)_i)$ for $1 \leq i \leq |S(v)|$, then Δ has a rule $q \to \ell(v)(q_1 \ldots q_{|S(v)|})$. We write $t \Longrightarrow_\rho q$ to denote that ρ is a run of A over t s.t. $\rho(\mathit{root}(t)) = q$. We use $t \Longrightarrow q$ to denote that $t \Longrightarrow_\rho q$ for some run ρ. The *language* of a state q is defined by $L(q) = \{t \mid t \Longrightarrow q\}$, and the *language* of A is defined by $L(A) = \bigcup_{q \in R} L(q)$.

Graphs and Forests with Ports. We will further work with graphs with designated input and output points. An *io-graph* is a pair (g, ϕ), abbreviated as g_ϕ, where g is a graph and $\phi \in \mathit{dom}(g)^+$ a sequence of *ports* in which ϕ_1 is the *input port* and $\phi_2 \cdots \phi_{|\phi|}$ is a sequence of *output ports* such that the occurrence of ports in ϕ is unique. Ports and joins of g are called *cut-points* of g_ϕ. We use $\mathit{cps}(g_\phi)$ to denote all cut-points of g_ϕ. We say that g_ϕ is *accessible* if it is accessible from the input port ϕ_1.

An io-forest is a pair $f = (t_1 \cdots t_n, \pi)$ s.t. $n \geq 1$ and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices, π_1 is the *input index*, and $\pi_2 \ldots \pi_{|\pi|}$ is a sequence of *output indices*, with no repetitions of indices in π. An io-forest encodes the io-graph $\otimes f$ where the ports of $\otimes t_1 \cdots t_n$ are roots of the trees defined by π, i.e., $\otimes f = (\otimes t_1 \cdots t_n, \mathit{root}(t_{\pi_1}) \cdots \mathit{root}(t_{\pi_n}))$.

Forest Automata. A *forest automaton* (FA) over Σ is a pair $F = (A_1 \cdots A_n, \pi)$ where $n \geq 1$, $A_1 \cdots A_n$ is a sequence of tree automata over $\Sigma \cup \{1, \ldots, n\}$, and $\pi \in \{1, \ldots, n\}^+$ is a sequence of port indices as defined for io-forests. The *forest language* of F is the set of io-forests $L_f(F) = L(A_1) \times \cdots \times L(A_n) \times \{\pi\}$, and the *graph language* of F is the set of io-graphs $L(F) = \{\otimes f \mid f \in L_f(F)\}$.

Structured Labels. We will further work with alphabets where symbols, called *structured labels*, have an inner structure. Let Γ be a ranked alphabet of *sub-labels*, ordered by a total ordering $\sqsubset_\Gamma$. We will work with graphs over the alphabet 2^Γ where for every symbol $A \subseteq \Gamma$, $\#A = \sum_{a \in A} \#a$. Let $e = v \mapsto (\{a_1, \ldots, a_m\}, v_1 \cdots v_n)$ be an edge of a graph g where $n = \sum_{1 \leq i \leq m} \#a_i$ and $a_1 \sqsubset_\Gamma a_2 \sqsubset_\Gamma \cdots \sqsubset_\Gamma a_m$. The triple $e\langle i \rangle = v \to (a_i, v_k \cdots v_l)$, $1 \leq i \leq m$, from the sequence $e\langle 1 \rangle = v \to (a_1, v_1 \cdots v_{\#a_1}), \ldots, e\langle m \rangle = v \to (a_m, v_{n-\#a_m+1} \cdots v_n)$ is called the i-th *sub-edge* of e (or the i-th sub-edge of v in g). We use $SE(g)$ to denote the set of all sub-edges of g. We say that a node v of a graph is *isolated* if it does not appear within any sub-edge, neither as an origin (i.e., $\ell(v) = \emptyset$) nor as a target. A graph g without isolated nodes is unambiguously determined by $SE(g)$ and vice versa (due to the total ordering $\sqsubset_\Gamma$ and since g has no isolated nodes). We further restrict ourselves to graphs with structured labels and without isolated nodes.

A counterpart of the notion of sub-edges in the context of rules of TA is the notion of rule-terms, defined as follows: Given a rule $\delta = (q, \{a_1, \ldots, a_m\}, q_1 \cdots q_n)$ of a TA over structured labels of 2^Γ, *rule-terms* of δ are the terms $\delta\langle 1 \rangle = a_1(q_1 \cdots q_{\#a_1}), \ldots, \delta\langle m \rangle = a_m(q_{n-\#a_m+1} \cdots q_n)$ where $\delta\langle i \rangle, 1 \leq i \leq m$, is called the *i-th rule-term of* δ.

Forest Automata of a Higher Level. We let Γ_1 be the set of all forest automata over 2^Γ and call its elements forest automata over Γ *of level 1*. For $i > 1$, we define Γ_i as the set of all forest automata over ranked alphabets $2^{\Gamma \cup \Delta}$ where $\Delta \subseteq \Gamma_{i-1}$ is any nonempty finite set of FA of level $i-1$. We denote elements of Γ_i as forest automata over Γ *of level* i. The rank $\#F$ of an FA F in these alphabets is the number of its output port indices. When used in an FA F over $2^{\Gamma \cup \Delta}$, the forest automata from Δ are called *boxes* of F. We write Γ_* to denote $\cup_{i \geq 0} \Gamma_i$ and assume that Γ_* is ordered by some total ordering $\sqsubset_{\Gamma_*}$.

An FA F of a higher level over Γ accepts graphs where forest automata of lower levels appear as sub-labels. To define the semantics of F as a set of graphs over Γ, we need the following operation of *sub-edge replacement* where a sub-edge of a graph is substituted by another graph. Intuitively, the sub-edge is removed, and its origin and targets are identified with the input and output ports of the substituted graph, respectively.

Formally, let g be a graph with an edge $e \in g$ and its i-th sub-edge $e\langle i \rangle = v_1 \to (a, v_2 \cdots v_n), 1 \leq i \leq |S_g(v_1)|$. Let g'_ϕ be an io-graph with $|\phi| = n$. Assume w.l.o.g. that $dom(g) \cap dom(g') = \emptyset$. The sub-edge $e\langle i \rangle$ can be replaced by g' provided that $\forall 1 \leq j \leq n : \ell_g(v_j) \cap \ell_{g'}(\phi_j) = \emptyset$, which means that the node $v_j \in dom(g)$ and the corresponding port $\phi_j \in dom(g')$ do not have successors reachable over the same symbol. If the replacement can be done, the result, denoted $g[g'_\phi / e\langle i \rangle]$, is the graph g_n in the sequence $g_0, \ldots, g_n$ of graphs defined as follows: $SE(g_0) = SE(g) \cup SE(g') \setminus \{e\langle i \rangle\}$, and for each $j : 1 \leq j \leq n$, the graph g_j arises from g_{j-1} by (1) deriving a graph h by replacing the origin of the sub-edges of the j-th port ϕ_j of g' by v_j, (2) redirecting edges leading to ϕ_j to v_j, i.e., replacing all occurrences of ϕ_j in $img(h)$ by v_j, and (3) removing ϕ_j.

If the symbol a above is an FA and $g'_\phi \in L(a)$, we say that $h = g[g'_\phi / e\langle i \rangle]$ is an *unfolding* of g, written $g \prec h$. Conversely, we say that g arises from h by *folding* g'_ϕ into

$e\langle i\rangle$. Let $\prec^*$ be the reflexive transitive closure of $\prec$. The Γ-*semantics* of g is then the set of graphs g' over Γ s.t. $g \prec^* g'$, denoted $[\![g]\!]_\Gamma$, or just $[\![g]\!]$ if no confusion may arise. For an FA F of a higher level over Γ, we let $[\![F]\!] = \bigcup_{g_\phi \in L(F)} ([\![g]\!] \times \{\phi\})$.

Canonicity. We call an io-forest $f = (t_1 \cdots t_n, \pi)$ *minimal* iff the roots of the trees $t_1 \cdots t_n$ are the cut-points of $\otimes f$. A minimal forest representation of a graph is unique up to reordering of $t_1 \cdots t_n$. Let the *canonical ordering* of cut-points of $\otimes f$ be defined by the cost of the cheapest paths leading from the input port to them. We say that f is *canonical* iff it is minimal, $\otimes f$ is accessible, and the trees within $t_1 \cdots t_n$ are ordered by the canonical ordering of their roots (which are cut-points of $\otimes f$). A canonical forest is thus a unique representation of an accessible io-graph. We say that an FA *respects canonicity* iff all forests from its forest language are canonical. Respecting canonicity makes it possible to efficiently test FA language inclusion by testing TA language inclusion of the respective components of two FA. This method is precise for FA of level 1 and sound (not always complete) for FA of a higher level [6].

In practice, we keep automata in the so called *state uniform* form, which simplifies maintaining of the canonicity respecting form [6] (and it is also useful when abstracting and "folding", as discussed in the following). It is defined as follows. Given a node v of a tree t in an io-forest, we define its *span* as the pair (α, V) where $\alpha \in \mathbb{N}^*$ is the sequence of labels of root references reachable from the root of t ordered according to the prices of the cheapest paths to them, and $V \subseteq \mathbb{N}$ is the set of labels of references which occur more than once in t. The state uniform form then requires that all nodes of forests from $L(F)$ that are labelled by the same state q in some accepting run of F have the same span, which we denote by $span(q)$.

3 FA-Based Shape Analysis

We now provide a high-level overview of the main loop of our shape analysis. The analysis automatically discovers memory safety errors (such as invalid dereferences of `null` or undefined pointers, double frees, or memory leaks) and provides an FA-represented over-approximation of the sets of heap configurations reachable at each program line. We consider sequential non-recursive C programs manipulating the heap. Each heap cell may have several *pointer selectors* and *data selectors* from some finite data domain (below, *PSel* denotes the set of pointer selectors, *DSel* denotes the set of data selectors, and $\mathbb{D}$ denotes the data domain).

Heap Representation. A single heap configuration is encoded as an io-graph $g_{\mathtt{sf}}$ over the ranked alphabet of structured labels 2^Γ with sub-labels from the ranked alphabet $\Gamma = PSel \cup (DSel \times \mathbb{D})$ with the ranking function that assigns each pointer selector 1 and each data selector 0. In this graph, an allocated memory cell is represented by a node v, and its internal structure of selectors is given by a label $\ell_g(v) \in 2^\Gamma$. Values of data selectors are stored directly in the structured label of a node as sub-labels from $DSel \times \mathbb{D}$, so, e.g., a singly linked list cell with the data value 42 and the successor node x_{next} may be represented by a node x such that $\ell_g(x) = \{\mathtt{next}(x_{next}), (\mathtt{data}, 42)(\varepsilon))\}$. Selectors with undefined values are represented such that the corresponding sub-labels are not in $\ell_g(x)$. The null value is modelled as the special node `null` such that $\ell_g(\mathtt{null}) = \emptyset$.

The input port sf represents a special node that contains the *stack frame* of the analysed function, i.e. a structure where selectors correspond to variables of the function.

In order to represent (infinite) *sets* of heap configurations, we use state uniform FA of a higher level to represent sets of canonical io-forests representing the heap configurations. The FA used as boxes are learnt during the analysis using the learning algorithm presented in Sec. 4.

Symbolic Execution. The verification procedure performs standard abstract interpretation with the abstract domain consisting of sets of state uniform FA (a single FA does not suffice as FA are not closed under union) representing sets of heap configurations at particular program locations. The computation starts from the initial heap configuration given by an FA for the io-graph $g_{\mathtt{sf}}$ where g comprises two nodes: null and sf where $\ell_g(\mathtt{sf}) = \emptyset$. The computation then executes abstract transformers corresponding to program statements until the sets of FA held at program locations stabilise. We note that abstract transformers corresponding to pointer manipulating statements are exact. Executing the abstract transformer $\tau_{\mathtt{op}}$ over a set of FA $\mathcal{S}$ is performed separately for every $F \in \mathcal{S}$. Some of boxes are first *unfolded* to uncover the accessed part of the heaps, then the update is performed. The detailed description of these steps can be found in [7].

At junctions of program paths, the analysis computes unions of sets of FA. At loop points, the union is followed by widening. The widening is performed by applying box *folding* and *abstraction* repeatedly in a loop on each FA from $\mathcal{S}$ until the result stabilises. An elaboration of these two operations, described in detail in Sec. 4 and 5 respectively, belongs to the main contribution of the presented paper.

4 Learning of Boxes

Sets of graphs with an unbounded number of joins can only be described by FA with the help of boxes. In particular, boxes allow one to replace (multiple) incoming sub-edges of a join by a single sub-edge, and hence lower the in-degree of the join. Decreasing the in-degree to 1 turns the join into an ordinary node. When a box is then used in a cycle of an FA, it effectively generates an unbounded number of joins.

The boxes are introduced by the operation of *folding* of an FA F which transforms F into an FA F' and a box B used in F' such that $[\![F]\!] = [\![F']\!]$. However, the graphs in $L(F')$ may contain less joins since some of them are hidden in the box B, which encodes a set of subgraphs containing a join and appearing repeatedly in the graphs of $L(F)$. Before we explain folding, we give a characterisation of subgraphs of graphs of $L(F)$ which we want to fold into a box B. Our choice of the subgraphs to be folded is a compromise between two high-level requirements. On the one hand, the folded subgraphs should contain incoming edges of joins and be as simple as possible in order to be reusable. On the other hand, the subgraphs should not be too small in order not to have to be subsequently folded within other boxes (in the worst case, leading to generation of unboundedly nested boxes). Ideally, the hierarchical structuring of boxes should respect the natural hierarchical structuring of the data structures being handled since if this is not the case, unboundedly many boxes may again be needed.

4.1 Knots of Graphs

A graph h is a *subgraph* of a graph g iff $SE(h) \subseteq SE(g)$. The *border* of h in g is the subset of the set $dom(h)$ of nodes of h that are incident with sub-edges in $SE(g) \setminus SE(h)$. A *trace* from a node u to a node v in a graph g is a set of sub-edges $t = \{e_0, \ldots, e_n\} \subseteq SE(g)$ such that $n \geq 1$, e_0 is an outgoing sub-edge of u, e_n is an incoming sub-edge of v, the origin of e_i is one of the targets of e_{i-1} for all $1 \leq i \leq n$, and no two sub-edges have the same origin. We call the origins of $e_1, \ldots, e_n$ the *inner nodes* of the trace. A trace from u to v is *straight* iff none of its inner nodes is a cut-point. A *cycle* is a trace from a node v to v. A *confluence* of g_ϕ is either a cycle of g_ϕ or it is the union of two disjoint traces starting at a node u, called the *base*, and ending in the node v, called the *tip* (for a cycle, the base and the tip coincide).

Given an io-graph g_ϕ, the *signature* of a sub-graph h of g is the minimum subset $sig(h)$ of $cps(g_\phi)$ that (1) contains $cps(g_\phi) \cap dom(h)$ and (2) all nodes of h, except the nodes of $sig(h)$ themselves, are reachable by straight traces from $sig(h)$. Intuitively, $sig(h)$ contains all cut-points of h plus the closest cut-points to h which lie outside of h but which are needed so that all nodes of h are reachable from the signature. Consider the example of the graph g_u in Fig. 2 in which cut-points are represented by $\bullet$. The signature of g_u is the set $\{u, v\}$. The signature of the highlighted subgraph h is also equal to $\{u, v\}$. Given a set $U \subseteq cps(g_\phi)$, a *confluence of* U is a confluence of g_ϕ with the signature within U. Intuitively, the confluence of a set of cut-points U is a confluence whose cut-points belong to U plus in case the base is not a cut-point, then the closest cut-point from which the base is reachable is also from U. Finally, the *closure* of U is the smallest subgraph h of g_ϕ that (1) contains all confluences of U and (2) for every inner node v of a straight trace of h, it contains all straight traces from v to leaves of g. The closure of the signature $\{u, v\}$ of the graph g_u in Fig. 2 is the highlighted subgraph h. Intuitively, Point 1 includes into the closure all nodes and sub-edges that appear on straight traces between nodes of U apart from those that do not lie on any confluence (such as node u in Fig. 2). Note that nodes x and y in Fig. 2, which are leaves of g_u, are not in the closure as they are not reachable from an inner node of any straight trace of h. The *closure of a subgraph* h of g_ϕ is the closure of its signature, and h is *closed* iff it equals its closure.

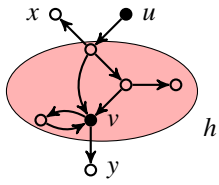


Fig. 2. Closure

Knots. For the rest of Sec. 4.1, let us fix an io-graph $g_\phi \in L(F)$. We now introduce the notion of a knot which summarises the desired properties of a subgraph k of g that is to be folded into a box. A *knot* k of g_ϕ is a subgraph of g such that: (1) k is a confluence, (2) k is the union of two knots with intersecting sets of sub-edges, or (3) k is the closure of a knot. A *decomposition* of a knot k is a set of knots such that the union of their sub-edges equals $SE(k)$. The *complexity of a decomposition* of k is the maximum of sizes of signatures of its elements. We define the *complexity of a knot* as the minimum of the complexities of its decompositions. A knot k of complexity n is an *optimal knot of complexity* n if it is maximal among knots of complexity n and if it has a root. The root must be reachable from the input port of g_ϕ by a trace that does not intersect with sub-edges of the optimal knot. Notice that the requirement of maximality implies that optimal knots are closed.

The following lemma, proven in [7], implies that optimal knots are uniquely identified by their signatures, which is crucial for the folding algorithm presented later.

Lemma 1. *The signature of an optimal knot of g_ϕ equals the signature of its closure.*

Next, we explain what is the motivation behind the notion of an optimal knot:

Confluences. As mentioned above, in order to allow one to eliminate a join, a knot must contain some join v together with at least one incoming sub-edge in case the knot is based on a loop and at least two sub-edges otherwise. Since g_ϕ is accessible (meaning that there do not exist any traces that cannot be extended to start from the same node), the edge must belong to some confluence c of g_ϕ. If the folding operation does not fold the entire c, then a new join is created on the border of the introduced box: one of its incoming sub-edges is labelled by the box that replaces the folded knot, another one is the last edge of one of the traces of c. Confluences are therefore the smallest subgraphs that can be folded in a meaningful way.

Uniting knots. If two different confluences c and c' share an edge, then after folding c, the resulting edge shares with c' two nodes (at least one being a target node), and thus c' contains a join of g_ϕ. To eliminate this join too, both confluences must be folded together. A similar reasoning may be repeated with knots in general. Usefulness of this rule may be illustrated by an example of the set of lists with head pointers. Without uniting, every list would generate a hierarchy of knots of the same depth as the length of the list, as illustrated in Fig. 3. This is clearly impractical since the entire set could not be represented using finitely many boxes. Rule 2 unites all knots into one that contains the entire list, and the set of all such knots can then be represented by a single FA (containing a loop accepting the inner nodes of the lists).

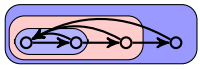


Fig. 3. A list with head pointers

Complexity of knots. The notion of complexity is introduced to limit the effect of Rule 2 of the definition of a knot, which unites knots that share a sub-edge, and to hopefully make it follow the natural hierarchical structuring of data structures. Consider, for instance, the case of singly-linked lists (SLLs) of cyclic doubly-linked lists (DLLs). In this case, it is natural to first fold the particular segments of the DLLs (denoted as DLSs below), i.e., to introduce a box for a single pair of next and prev pointers. This way, one effectively obtains SLLs of cyclic SLLs. Subsequently, one can fold the cyclic SLLs into a higher-level box. However, uniting all knots with a common sub-edge would create knots that contain entire cyclic DLLs (requiring unboundedly many joins inside the box). The reason is that in addition to the confluences corresponding to DLSs, there are confluences which traverse the entire cyclic DLLs and that share sub-edges with all DLSs (this is in particular the case of the two circular sequences consisting solely of next and prev pointers respectively). To avoid the undesirable folding, we exploit the notion of complexity and fold graphs in successive rounds. In each round we fold all optimal knots with the smallest complexity (as described in Sec. 4.2), which should correspond to the currently most nested, not yet folded, sub-structures. In the previous example, the algorithm starts by folding DLSs of complexity 2, because the complexity of the confluences in cyclic DLLs is given by the number of the DLSs they traverse.

Closure of knots. The closure is introduced for practical reasons. It allows one to identify optimal knots by their signatures, which is then used to simplify automata constructions that implement folding on the level of FA (cf. Sec. 4.2).

Root of an optimal knot. The requirement for an optimal knot k to have a root is to guarantee that if an io-graph h_Ψ containing a box B representing k is accessible, then the io-graph $h_\Psi[k/B]$ emerging by substituting k for a sub-edge labelled with B is accessible, and vice versa. It is also a necessary condition for the existence of a canonical forest representation of the knot itself (since one needs to order the cut-points w.r.t. the prices of the paths leading to them from the input port of the knot).

4.2 Folding in the Abstraction Loop

In this section, we describe the operation of folding together with the main abstraction loop of which folding is an integral part. The pseudo-code of the main abstraction loop is shown in Alg. 1. The algorithm modifies a set of FA until it reaches a fixpoint. Folding on line 5 is a sub-procedure of the algorithm which looks for substructures of FA that accept optimal knots, and replaces these substructures by boxes that represent the corresponding optimal knots. The operation of folding is itself composed of four consecutive steps: *Identifying indices*, *Splitting*, *Constructing boxes*, and *Applying boxes*. For space reasons, we give only an overview of the steps of the main abstraction loop and folding. Details may be found in [7].

```
1 Unfold solitaire boxes
2 repeat
3     Normalise
4     Abstract
5     Fold
6 until fixpoint
```

Alg. 1: Abstraction Loop

Unfolding of Solitaire Boxes. Folding is in practice applied on FA that accept partially folded graphs (only some of the optimal knots are folded). This may lead the algorithm to hierarchically fold data structures that are not hierarchical, causing the symbolic execution not to terminate. For example, consider a program that creates a DLL of an arbitrary length. Whenever a new DLS is attached, the folding algorithm would enclose it into a box together with the tail which was folded previously. This would lead to creation of a hierarchical structure of an unbounded depth (see Fig. 4), which would cause the symbolic execution to never reach a fixpoint. Intuitively, this is a situation when a repetition of subgraphs may be expressed by an automaton loop that iterates a box, but it is instead misinterpreted as a recursive nesting of graphs. This situation may happen when a newly created box contains another box that cannot be iterated since it does not appear on a loop (e.g, in Fig. 4 there is always one occurrence of a box encoding a shorter DLL fragment inside a higher-level box). This issue is addressed in the presented algorithm by first unfolding all occurrences of boxes that are not iterated by automata loops before folding is started.

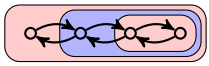


Fig. 4. DLL

Normalising. We define the *index* of a cut-point $u \in cps(g_\phi)$ as its position in the canonical ordering of cut-points of g_ϕ, and the *index* of a closed subgraph h of g_ϕ as the set of indices of the cut-points in $sig(h)$. The folding algorithm expects the input FA F to satisfy the property that all io-graphs of $L(F)$ have the same indices of closed knots. The reason is that folding starts by identifying the index of an optimal knot of an arbitrary io-graph from $L(F)$, and then it creates a box which accepts all closed subgraphs of the io-graphs from g_ϕ with the same index. We need a guarantee that *all* these subgraphs are indeed optimal knots. This guarantee can be achieved if the io-graphs from $L(F)$ have equivalent interconnections of cut-points, as defined below.

We define the relation $\sim_{g_\phi} \subseteq \mathbb{N} \times \mathbb{N}$ between indices of closed knots of g_ϕ such that $N \sim_{g_\phi} N'$ iff there is a closed knot k of g_ϕ with the index N and a closed knot k' with the index N' such that k and k' have intersecting sets of sub-edges. We say that two io-graphs g_ϕ and h_ψ are *interconnection equivalent* iff $\sim_{g_\phi} = \sim_{h_\psi}$.

Lemma 2. *Interconnection equivalent io-graphs have the same indices of optimal knots.*

Interconnection equivalence of all io-graphs in the language of an FA F is achieved by transforming F to the *interconnection respecting form*. This form requires that the language of every TA of the FA consists of interconnection equivalent trees (when viewing root references and roots as cut-points with corresponding indices). The transformation is described in [7]. The normalisation step also includes a transformation into the state uniform and canonicity respecting form.

Abstraction. We use abstraction described in Sec. 5 that preserves the canonicity respecting form of TA as well as their state uniformity. It may break interconnection uniformity, in which case it is followed by another round of normalisation. Abstraction is included into each round of folding for the reason that it leads to learning more general boxes. For instance, an FA encoding a cyclic list of one particular length is first abstracted into an FA encoding a set of cyclic lists of all lengths, and the entire set is then folded into a single box.

Identifying Indices. For every FA F entering this sub-procedure, we pick an arbitrary io-graph $g_\phi \in L(F)$, find all its optimal knots of the smallest possible complexity n, and extract their indices. By Lemma 2 and since F is normalised, indices of the optimal knots are the same for all io-graphs in $L(F)$. For every found index, the following steps fold all optimal knots with that index at once. Optimal knots of complexity n do not share sub-edges, the order in which they are folded is therefore not important.

Splitting. For an FA $F = (A_1 \cdots A_n, \pi)$ and an index I of an optimal knot found in the previous step, splitting transforms F into a (set of) new FA with the same language. The nodes of the borders of I-indexed optimal knots of io-graphs from $L(F)$ become roots of trees of io-forests accepted by the new FA. Let $s \in I$ be a position in F such that the s-indexed cut-points of io-graphs from $L(F)$ reach all the other I-indexed cut-points. The index s exists since an optimal knot has a root. Due to the definition of the closure, the border contains all I-indexed cut-points, with the possible exception of s. The s-th cut-point may be replaced in the border of the I-indexed optimal knot by the base e of the I-indexed confluence that is the first one reached from the s-th cut-point by a straight path. We call e the *entry*. The entry e is a root of the optimal knot, and the s-th cut-point is the only I-indexed cut-point that might be outside the knot. If e is indeed different from the s-th cut-point, then the s-th tree of forests accepted by F must be split into two trees in the new FA: The subtree rooted at the entry is replaced by a reference to a new tree. The new tree then equals the subtree of the original s-th tree rooted at the entry.

The construction is carried out as follows. We find all states and all of their rules that accept entry nodes. We denote such states and rules as entry states and rules. For every entry state q, we create a new FA F_q^0 which is a copy of F but with the s-th TA A_s split to a new s-th TA A'_s and a new $(n+1)$-th TA A_{n+1}. The TA A'_s is obtained from A_s by changing the entry rules of q to accept just a reference to the new $(n+1)$-th root and by removing entry rules of all other entry states (the entry states are processed separately in

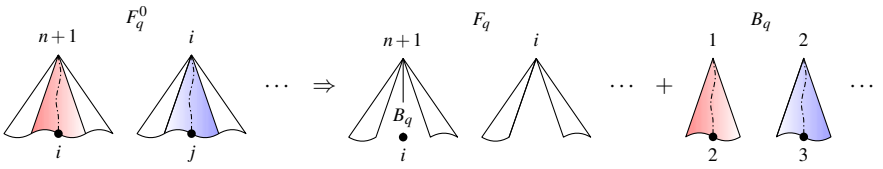


Fig. 5. Creation of F_q and B_q from F_q^0. The subtrees that contain references $i, j \in J$ are taken into B_q, and replaced by the B_q-labelled sub-edge in F_q.

order to preserve possibly different contexts of entry nodes accepted at different states). The new TA A_{n+1} is a copy of A_s but with the only accepting state being q. Note that the construction is justified since due to state uniformity, each node that is accepted by an entry rule and that does not appear below a node that is also accepted by an entry rule is an entry node. In the result, the set $J = (I \setminus \{s\}) \cup \{n+1\}$ contains the positions of the trees of forests of F_q^0 rooted at the nodes of the borders of I-indexed optimal knots.

Constructing Boxes. For every F_q^0 and J being the result of splitting F according to an index I, a box B_q is constructed from F_q^0. We transform TA of F_q^0 indexed by the elements of J. The resulting TA will accept the original trees up to that the roots are stripped from the children that cannot reach a reference to J. To turn these TA into an FA accepting optimal knots with the index I, it remains to order the obtained TA and define port indices, which is described in detail in [7]. Roughly, the input index of the box will be the position j to which we place the modified $(n+1)$-th TA of F_q^0 (the one that accepts trees rooted at the entry). The output indices are the positions of the TA with indices $J \setminus \{j\}$ in F_q^0 which accept trees rooted at cut-points of the border of the optimal knots.

Applying Boxes. This is the last step of folding. For every F_q^0, J, and B_q which are the result of splitting F according to an index I, we construct an FA F_q that accepts graphs of F where knots enclosed in B_q are substituted by a sub-edge with the label B_q. It is created from F_q^0 by (1) leaving out the parts of root rules of its TA that were taken into B_q, and (2) adding the rule-term $B_q(r_1, \ldots, r_m)$ to the rule-terms of root rules of the $(n+1)$-th component of F_q^0 (these are rules used to accept the roots of the optimal knots enclosed in B_q). The states $r_1, \ldots, r_m$ are fresh states that accept root references to the appropriate elements of J (to connect the borders of knots of B_q correctly to the graphs of F_q—the details may be found in [7]). The FA F_q now accepts graphs where optimal knots of graphs of $L(F)$ with the signature I are hidden inside B_q. Creation of B_q and of its counterpart F_q from F_q^0 is illustrated in Fig. 5 where $i, j, \ldots \in J$.

During the analysis, the discovered boxes must be stored in a database and tested for equivalence with the newly discovered ones since the alphabets of FA would otherwise grow with every operation of folding *ad infinitum*. That is, every discovered box is given a unique name, and whenever a semantically equivalent box is folded, the newly created edge-term is labelled by that name. This step offers an opportunity for introducing another form of acceleration of the symbolic computation. Namely, when a box B is found by the procedure described above, and another box B' with a name N s.t. $[\![B']\!] \subset [\![B]\!]$ is already in the database, we associate the name N with B instead of with B' and restart the analysis (i.e., start the analysis from the scratch, remembering just the updated database of boxes). If, on the other hand, $[\![B]\!] \subseteq [\![B']\!]$, the folding is performed using the name N

of B', thus overapproximating the semantics of the folded FA. As presented in Sec. 6, this variant of the procedure, called *folding by inclusion*, performs in some difficult cases significantly better than the former variant, called *folding by equivalence*.

5 Abstraction

The abstraction we use in our analysis is based on the general techniques described in the framework of abstract regular (tree) model checking [2]. We, in particular, build on the *finite height abstraction* of TA. It is parameterised by a height $k \in \mathbb{N}$, and it collapses TA states q, q' iff they accept trees with the same sets of prefixes of the height at most k (the prefix of height k of a tree is a subgraph of the tree which contains all paths from the root of length at most k). This defines an equivalence on states denoted by $\approx_k$. The equivalence $\approx_k$ is further refined to deal with various features special for FA. Namely, it has to work over tuples of TA and cope with the interconnection of the TA via root references, with the hierarchical structuring, and with the fact that we use a *set* of FA instead of a single FA to represent the abstract context at a particular program location.

Refinements of $\approx_k$. First, in order to maintain the same basic shape of the heap after abstraction (such that no cut-point would, e.g., suddenly appear or disappear), we refine $\approx_k$ by requiring that equivalent states must have the same spans (as defined in Sec. 2). When applied on $\approx_1$, which corresponds to equivalence of data types, this refinement provided enough precision for most of the case studies presented later on, with the exception of the most difficult ones, namely programs with skip lists [13]. To verify these programs, we needed to further refine the abstraction to distinguish automata states whenever trees from their languages encode tree components containing a different number of unique paths to some root reference, but some of these paths are hidden inside boxes. In particular, two states q, q' can be equivalent only if for every io-graph g_ϕ from the graph language of the FA, for every two nodes $u, v \in dom(g_\phi)$ accepted by q and q', respectively, in an accepting run of the corresponding TA, the following holds: For every $w \in cps(g_\phi)$, both u and v have the same number of outgoing sub-edges (selectors) in $[\![g_\phi]\!]$ which start a trace in $[\![g_\phi]\!]$ leading to w. According to our experiments, this refinement does not cost almost any performance, and hence we use it by default.

Abstraction for Sets of FA. Our analysis works with sets of FA. We observed that abstracting individual FA from a set of FA in isolation is sometimes slow since in each of the FA, the abstraction widens some selector paths only, and it takes a while until an FA in which all possible selector paths are widened is obtained. For instance, when analysing a program that creates binary trees, before reaching a fixpoint, the symbolic analysis generates many FA, each of them accepting a subset of binary trees with some of the branches restricted to a bounded length (e.g., trees with no right branches, trees with a single right branch of length 1, length 2, etc.). In such cases, it helps when the abstraction has an opportunity to combine information from several FA. For instance, consider an FA that encodes binary trees degenerated to an arbitrarily long left branch, and another FA that encodes trees degenerated to right branches only. Abstracting these FA in isolation has no effect. However, if the abstraction is allowed to collapse states from both of these FA, it can generate an FA accepting all possible branches.

Unfortunately, the natural solution to achieve the above, which is to unite FA before abstraction, cannot be used since FA are not closed under union (uniting TA component-

wise overapproximates the union). However, it is possible to enrich the automata structure of an FA F by TA states and rules of another one without changing the language of F, and in this way allow the abstraction to combine the information from both FA. In particular, before abstracting an FA $F = (A_1 \cdots A_n, \pi)$ from a set S of FA, we pre-process it as follows. (1) We pick automata $F' = (A'_1 \cdots A'_n, \pi) \in S$ which are compatible with F in that they have the same number of TA, the same port references, and for each $1 \leq i \leq n$, the root states of A'_i have the same spans as the root states of A_i. (2) For all such F' and each $1 \leq i \leq n$, we add rules and states of A'_i to A_i, but we keep the original set of root states of A_i. Since we assume that the sets of state of TAs of different FA are disjoint, the language of A_i stays the same, but its structure is enriched, which helps the abstraction to perform a coarser widening.

6 Experimental Results

We have implemented the above proposed techniques in the Forester tool and tested their generality and efficiency on a number of case studies. In the experiments, we compare two configurations of Forester, and we also compare the results of Forester with those of Predator [4], which uses a graph-based memory representation inspired by separation logic with higher-order list predicates. We do not provide a comparison with Space Invader [12] and SLAyer [1], based also on separation logic with higher-order list predicates, since in our experiments they were outperformed by Predator.

In the experiments, we considered programs with various types of lists (singly and doubly linked, cyclic, nested, with skip pointers), trees, and their combinations. In the case of skip lists, we had to slightly modify the algorithms since their original versions use an ordering on the data stored in the nodes of the lists (which we currently do not support) in order to guarantee that the search window delimited on some level of skip pointers is not left on any lower level of the skip pointers. In our modification, we added an additional explicit end-of-window pointer. We checked the programs for memory safety only, i.e., we did not check data-dependent properties.

Table 1 gives running times in seconds (the average of 10 executions) of the tools on our case studies. “Basic” stands for Forester with the abstraction applied on individual FA only and “SFA” stands for Forester with the abstraction for sets of FA. The value T means that the running time of the tool exceeded 30 minutes, and the value Err means that the tool reported a spurious error. The names of the examples in the table contain the name of the data structure manipulated in the program, which is “SLL” for singly linked lists, “DLL” for doubly linked lists (the “C” prefix denotes cyclic lists), “tree” for binary trees, “tree+parents” for trees with parent pointers. Nested variants of SLL (DLL) are named as “SLL (DLL) of” and the type of the nested structure. In particular, “SLL of 0/1 SLLs” stands for SLL of a nested SLL of length 0 or 1, and “SLL of 2CDLLs” stands for SLL whose each node is a root of two CDLLs. The “+head” flag stands for a list where each element points to the head of the list and the subscript “Linux” denotes the implementation of lists used in the Linux kernel, which uses type casts and a restricted pointer arithmetic. The “DLL+subdata” stands for a kind of a DLL with data pointers pointing either inside the list nodes or optionally outside of them. For a “skip list”, the subscript denotes the number of skip pointers. In the example “tree+stack”, a

Table 1. Results of the experiments

Example	basic	SFA	boxes	Predator	Example	basic	SFA	boxes	Predator
SLL (delete)	0.03	0.04		0.04	DLL (reverse)	0.04	0.06	1 / 1	0.03
SLL (bubblesort)	0.04	0.04		0.03	DLL (insert)	0.06	0.07	1 / 1	0.05
SLL (mergesort)	0.08	0.15		0.10	DLL (insertsort1)	0.35	0.40	1 / 1	0.11
SLL (insertsort)	0.05	0.05		0.04	DLL (insertsort2)	0.11	0.12	1 / 1	0.05
SLL (reverse)	0.03	0.03		0.03	DLL of CDLLs	5.67	1.25	8 / 7	0.22
SLL+head	0.05	0.05		0.03	DLL+subdata	0.06	0.09	- / 2	T
SLL of 0/1 SLLs	0.03	0.03		0.11	CDLL	0.03	0.03	1 / 1	0.03
SLL_{Linux}	0.03	0.03		0.03	tree	0.14	0.14		Err
SLL of CSLLs	2.07	0.73	3 / 4	0.12	tree+parents	0.18	0.21	2 / 2	T
SLL of $2CDLLs_{Linux}$	0.16	0.17	13 / 5	0.25	tree+stack	0.09	0.08		Err
skip $list_2$	0.66	0.42	- / 3	T	tree (DSW)	1.74	0.40		Err
skip $list_3$	T	9.14	- / 7	T	tree of CSLLs	0.32	0.42	- / 4	Err

randomly constructed tree is deleted using a stack, and "DSW" stands for the Deutsch-Schorr-Waite tree traversal (the Lindstrom variant). All experiments start with a random creation and end with a disposal of the specified structure while the indicated procedure (if any) is performed in between. The experiments were run on a machine with the Intel i7-2600 (3.40 GHz) CPU and 16 GiB of RAM.

The table further contains the column "boxes" where the value "X/Y" means that X manually created boxes were provided to the analysis that did not use learning while Y boxes were learnt when the box learning procedure was enabled. The value "-" of X means that we did not run the given example with manually constructed boxes since their construction was too tedious. If user-defined boxes are given to Forester in advance, the speedup is in most cases negligible, with the exception of "DLL of CDLLs" and "SLL of CSLLs", where it is up to 7 times. In a majority of cases, the learnt boxes were the same as the ones created manually. However, in some cases, such as "SLL of $2CDLLs_{Linux}$", the learning algorithm found a smaller set of more elaborate boxes than those provided manually.

In the experiments, we use folding by inclusion as defined in Sec. 4.2. For simpler cases, the performance matched the performance of folding by equivalence, but for the more difficult examples it was considerably faster (such as for "skip $list_2$" when the time decreased from 3.82 s to 0.66 s), and only when it was used the analysis of "skip $list_3$" succeeded. Further, the implementation folds optimal knots of the complexity ≤ 2 which is enough for the considered examples. Finally, note that the performance of Forester in the considered experiments is indeed comparable with that of Predator even though Forester can handle much more general data structures.

7 Conclusion

We have proposed a new shape analysis using forest automata which—unlike the previously known approach based on FA—is fully automated. For that purpose, we have proposed a technique of automatically learning FA called boxes to be used as alphabet symbols in higher-level FA when describing sets of complex heap graphs. We have also proposed a way how to efficiently integrate the learning with the main analysis

algorithm. Finally, we have proposed a significant improvement—both in terms of generality as well as efficiency—of the abstraction used in the framework. An implementation of the approach in the Forester tool allowed us to fully-automatically handle programs over quite complex heap structures, including 2-level and 3-level skip lists, which—to the best of our knowledge—no other fully-automated verification tool can handle. At the same time, the efficiency of the analysis is comparable with other state-of-the-art analysers even though they handle less general classes of heap structures.

For the future, there are many possible ways how the presented approach can be further extended. First, one can think of using recursive boxes or forest automata using hedge automata as their components in order to handle even more complex data structures (such as mcf trees). Another interesting direction is that of integrating FA-based heap analysis with some analyses for dealing with infinite non-pointer data domains (e.g., integers) or parallelism.

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306, 13-37876P), the Czech Ministry of Education, Youth, and Sports (project MSM 0021630528), the BUT FIT project FIT-S-12-1, and the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070.

References

1. Berdine, J., Cook, B., Ishtiaq, S.: Memory Safety for Systems-level Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
2. Bouajjani, A., Habermehl, P., Rogalewicz, A., Vojnar, T.: Abstract Regular (Tree) Model Checking. STTT 14(2) (2012)
3. Chang, B.-Y.E., Rival, X., Necula, G.C.: Shape Analysis with Structural Invariant Checkers. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 384–401. Springer, Heidelberg (2007)
4. Dudka, K., Peringer, P., Vojnar, T.: Predator: A Practical Tool for Checking Manipulation of Dynamic Data Structures Using Separation Logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
5. Guo, B., Vachharajani, N., August, D.I.: Shape Analysis with Inductive Recursion Synthesis. In: Proc. of PLDI 2007. ACM Press (2007)
6. Habermehl, P., Holík, L., Rogalewicz, A., Šimáček, J., Vojnar, T.: Forest Automata for Verification of Heap Manipulation. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 424–440. Springer, Heidelberg (2011)
7. Holík, L., Lengál, O., Rogalewicz, A., Šimáček, J., Vojnar, T.: Fully Automated Shape Analysis Based on Forest Automata. Tech. rep. FIT-TR-2013-01, FIT BUT (2013)
8. Heinen, J., Noll, T., Rieger, S.: Juggrnaut: Graph Grammar Abstraction for Unbounded Heap Structures. ENTCS 266 (2010)
9. Lee, O., Yang, H., Petersen, R.: Program Analysis for Overlaid Data Structures. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 592–608. Springer, Heidelberg (2011)
10. Magill, S., Tsai, M.-H., Lee, P., Tsay, Y.-K.: Automatic Numeric Abstractions for Heap-manipulating programs. In: Proc. of POPL 2010. ACM Press (2010)
11. Weinert, A.D.: Inferring Heap Abstraction Grammars. BSc thesis, RWTH Aachen (2012)
12. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable Shape Analysis for Systems Code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
13. Pugh, W.: Skip Lists: A Probabilistic Alternative to Balanced Trees. Commun. ACM 33(6), 668–676 (1990)

Effectively-Propositional Reasoning about Reachability in Linked Data Structures*

Shachar Itzhaky[1], Anindya Banerjee[2], Neil Immerman[3], Aleksandar Nanevski[2], and Mooly Sagiv[1]

[1] Tel Aviv University, Tel Aviv, Israel
[2] IMDEA Software Institute, Madrid, Spain
[3] University of Massachusetts, Amherst, USA

Abstract. This paper proposes a novel method of harnessing existing SAT solvers to verify reachability properties of programs that manipulate linked-list data structures. Such properties are essential for proving program termination, correctness of data structure invariants, and other safety properties. Our solution is complete, i.e., a SAT solver produces a counterexample whenever a program does not satisfy its specification. This result is surprising since even first-order theorem provers usually cannot deal with reachability in a complete way, because doing so requires reasoning about transitive closure.

Our result is based on the following ideas: (1) Programmers must write assertions in a restricted logic without quantifier alternation or function symbols. (2) The correctness of many programs can be expressed in such restricted logics, although we explain the tradeoffs. (3) Recent results in descriptive complexity can be utilized to show that every program that manipulates potentially cyclic, singly- and doubly-linked lists and that is annotated with assertions written in this restricted logic, can be verified with a SAT solver.

We implemented a tool atop Z3 and used it to show the correctness of several linked list programs.

1 Introduction

This paper shows that it is possible to reason about reachability between dynamically allocated memory locations in potentially cyclic, singly-linked and doubly-linked lists using effectively-propositional reasoning. We present a novel method that can harness existing SAT solvers to verify reachability properties of programs that manipulate linked-list data structures, and to produce a concrete counterexample whenever a program does not satisfy its specification. This result is surprising because the natural specification of such programs involves quantifiers, inductive definitions and transitive closure, thus

* Itzhaky and Sagiv were funded by the European Research Council under the European Union's Seventh Framework Program (FP7/2007-2013) / ERC grant agreement no. [321174-VSSC] and by a grant from the Israel Science Foundation (652/11). Banerjee and Nanevski were partially supported by by Spanish MINECO projects TIN2009-14599-C03-02 Desafios, TIN2010-20639 Paran10, TIN2012-39391-C04-01 Strongsoft, EU NoE Project 256980 Nessos, AMAROUT grant PCOFUND-GA-2008-229599, and Ramon y Cajal grant RYC-2010-0743. Immerman was partially supported by NSF grant CCF 1115448.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 756–772, 2013.

precluding first-order, automatic theorem provers from dealing with reachability in a complete way.

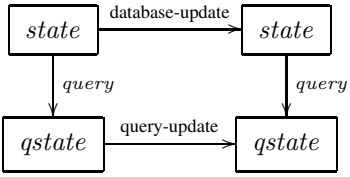


Fig. 1. The view update problem. Queries are expressed by formulas in a rich logic with transitive closure, but query-update is expressed essentially propositionally.

Two central observations underpin our method. (i) In programs that manipulate singly- and doubly-linked lists it is possible to express the 'next' pointer in terms of the reachability relation between list elements. This permits direct use of recent results in descriptive complexity [10]: we can maintain reachability with respect to heap mutation in a precise manner. Moreover, we can axiomatize reachability using quantifier free formulas. (ii) In order to handle statements which traverse the heap, we allow verification conditions (VCs) with $\forall^*\exists^*$ formulas so that they can be discharged by SAT solvers (as we explain shortly). However, we allow the programmer to only write assertions in a restricted fragment of FOL that disallows formulas with quantifier alternations but allows reflexive transitive closure. The main reason is that invariants occur both in the antecedent and in the consequent of the VC for loops; thus the assertion language has to be closed under negation.

The appeal to descriptive complexity stems from the fact that recently it has been applied to the view-update problem in databases. This problem has a pleasant parallel to the heap reachability update problem we are considering. In the view-update problem, the logical complexity of updating a query wrt. database modifications is lower than computing the query for the updated database from scratch (depicted in Fig. 1). Indeed, the latter uses formulas with transitive closure, while the former uses quantifier-free formulas without transitive closure. In our setting, we compute reachability relations instead of queries. We exploit the fact that the logical complexity of adapting the (old) reachability relation to the updated heap is lower than computing the new reachability relation from scratch. The solution we employ is similar to the use of dynamic graph algorithms for solving the view-update problem, where directed paths between nodes are updated when edges are added/removed (e.g., see [5]), except that our solution is geared towards verification of heap-manipulating programs with linked data structures.

Main Results

– We define AF^R, a new logic for expressing properties of programs, that is an *alternation free* sub-fragment of FO^{TC} (i.e., first-order logic with transitive closure): alternation between universal and existential quantifiers in formulas is disallowed. A distinguishing feature of AF^R is that it allows relation symbols but does not allow direct application of function symbols. Atomic formulas of AF^R may denote reachability relations between memory locations via pointers such as $next$ and $prev$ fields in linked lists, or any other relations without transitive closure.
– We empirically show that loop invariants in many programs manipulating singly- and doubly-linked lists can be specified using AF^R formulas.

– We show that the effect of many procedures manipulating singly- and doubly-linked lists can be specified using AF^R formulas. This result may require that the memory that the procedure manipulates be "owned" by its formal parameters.
– We show direct use of existing results in dynamic complexity [10] to prove that AF^R formulas are closed under weakest preconditions for statements which destructively update memory (e.g., $x.next := y$).
– For statements that traverse the heap (e.g., $x := y.next$), AF^R formulas are *not* closed under weakest preconditions. For these cases we show that weakest preconditions are expressible in the AE^R logic which generalizes AF^R by permitting existential quantification inside universal quantification. AE^R formulas are decidable for validity since their negation has the form $\exists^*\forall^*$, and fits in the Bernays-Schönfinkel fragment which is decidable for satisfiability [19]. In fact, they can be checked with a SAT solver by replacing existential quantifiers with constants, and universal quantifications by conjunctions over the constants. Indeed, Z3 [4] is complete for these formulas.
– We report on experiments with a tool that checks correctness of several, commonly used heap-manipulating structured programs, and that uses Z3 as back-end. The tool can determine whether or not program annotations (pre- and postconditions, loop invariants) are AF^R formulas, and can check both safety and equivalence of procedures. The tool is sound and also complete in the sense that it generates concrete counterexamples for programs violating the VCs.

This paper is accompanied by a technical report containing further examples and proofs.

2 Overview

2.1 Programming with Restricted Invariants

In this paper we require that the specified invariants are AF^R formulas. That is, they only use reflexive transitive closure but do not explicitly use function symbols and quantifier alternations.

Definition 1. *Let $t_1, t_2, \ldots t_n$ be logical variables or constant symbols. We define four types of* ***atomic propositions****: (i) $t_1 = t_2$ denoting equality, (ii) $r(t_1, t_2, \ldots, t_n)$ denoting the application of relation symbol r of arity n, and (iii) $t_1\langle f^*\rangle t_2$ denoting the existence of $k \geq 0$ such that $f^k(t_1) = t_2$, where $f^0(t_1) \stackrel{\text{def}}{=} t_1$, and $f^{k+1}(t_1) \stackrel{\text{def}}{=} f(f^k(t_1))$. We say that $t_1\langle f^*\rangle t_2$ is a* ***reachability constraint*** *between t_1 and t_2 via the function f.* ***Quantifier-free formulas*** *(QF^R) are Boolean combinations of such formulas without quantifiers.* ***Alternation-free formulas*** *(AF^R) are Boolean combinations of such formulas with additional quantifiers of the form $\forall^*{:}\varphi$ or $\exists^*{:}\varphi$ where φ is a QF^R formula.* ***Forall-Exists Formulas*** *(AE^R) formulas are Boolean combinations of such formulas with additional quantifiers of the form $\forall^*\exists^*{:}\varphi$ where φ is a QF^R formula. In particular, $QF^R \subset AF^R \subset AE^R$.*

Fig. 2 presents a Java program for in-situ reversal of a linked list. Every node of the list has a *next* field that points to its successor node in the list. Thus, we can model *next* as a function that maps a node in the list to its successor. For simplicity we assume that the

program manipulates the entire heap, that is, the heap consists of just the nodes in the linked list. To describe the heap that is reachable from the formal parameter h, where h points to the head of the input list, we use the formula $\forall\alpha : h\langle \mathit{next}^*\rangle\alpha$.

We also assume, until Section 5, that the heap is acyclic, i.e., the formula ac below is a precondition of $\mathit{reverse}$.

$$ac \stackrel{\text{def}}{=} \forall\alpha,\beta : \alpha\langle \mathit{next}^*\rangle\beta \wedge \beta\langle \mathit{next}^*\rangle\alpha \rightarrow \alpha = \beta \tag{1}$$

Table 1. AF^R invariants for $\mathit{reverse}$. Note that *next*, next_0 are function symbols while $\alpha\langle \mathit{next}^*\rangle\beta$, $\alpha\langle \mathit{next}_0^*\rangle\beta$ are atomic propositions on the reachability via directed paths from α to β consisting of *next*, next_0 edges.

$$I_0 \stackrel{\text{def}}{=} ac \wedge \forall\alpha : h\langle \mathit{next}^*\rangle\alpha$$

$$I_3 \stackrel{\text{def}}{=} ac \wedge \forall\alpha,\beta \neq \mathit{null} : \left\{ \begin{array}{ll} \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha & d\langle \mathit{next}^*\rangle\alpha \\ c\langle \mathit{next}^*\rangle\alpha \wedge (\alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \alpha\langle \mathit{next}_0^*\rangle\beta) & \neg d\langle \mathit{next}^*\rangle\alpha \end{array} \right\}$$

$$I_9 = ac \wedge \forall\alpha : d\langle \mathit{next}^*\rangle\alpha \wedge (\forall\alpha,\beta : \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha)$$

```
Node reverse(Node h) {
  0: Node c = h;
  1: Node d = null;
  2: while  3: (c != null) {
       4: Node t = c.next;
       5: c.next = null;
       6: c.next = d;
       7: d = c;
       8: c = t;
  }
  9: return d;
}
```

Fig. 2. A simple Java program that reverses a list in-situ

Table 1 shows the invariants I_0, I_3 and I_9 that describe a precondition, a loop invariant, and a postcondition of $\mathit{reverse}$. They are expressed in AF^R which permits use of function symbols (e.g. *next*) in formulas only to express reachability (cf. next^*); moreover, quantifier alternation is not permitted. The notation $\left\{ \begin{array}{ll} f & b \\ g & \neg b \end{array} \right\}$ is shorthand for the conditional $(b \wedge f) \vee (\neg b \wedge g)$.

Note that I_3 and I_9 refer to next_0, the value of *next* at procedure entry. The postcondition I_9 says that $\mathit{reverse}$ preserves acyclicity of the list and updates *next* so that, upon procedure termination, the links of the original list have been reversed. It also says that all the nodes are reachable from d in the reversed list. I_3 says that at loop entry c is non-null and moreover, the original list is partially reversed. That is, any node reachable from d is connected in reverse wrt. the input list, whereas any node not reachable from d is reachable from c and belongs to the part of the list that has not yet been reversed. Observe that I_3 and I_9 only refer to next^* and never to *next* alone. A more natural way to express I_9 would be

$$I_9' \stackrel{\text{def}}{=} ac \wedge \forall\alpha : d\langle \mathit{next}^*\rangle\alpha \wedge (\forall\alpha,\beta : \mathit{next}(\alpha) = \beta \Leftrightarrow \mathit{next}_0(\beta) = \alpha) \tag{2}$$

But this formula is not in AF^R because it explicitly refers to function symbols *next* and next_0 outside a reachability constraint.

2.2 Inverting Reachability Constraints

A crucial step in moving from arbitrary FO^{TC} formulas to AF^R formulas is eliminating explicit uses of functions such as *next*. While this may be difficult for a general graph, we show that this can be done for programs that manipulate (potentially cyclic) singly- and doubly-linked lists. In this section, we informally demonstrate this elimination for acyclic lists. We observe that if *next* is acyclic, we can construct $next^+$ from $next^*$ by

$$\alpha\langle next^+\rangle\beta \Leftrightarrow \alpha\langle next^*\rangle\beta \wedge \alpha \neq \beta \tag{3}$$

Also, since *next* is a function, the set of nodes reachable from a node α is totally ordered by $next^*$. Therefore, $next(\alpha)$ is the minimal node in this order that is not α. The minimality is expressed using extra universal quantification in

$$next(\alpha) = \beta \Leftrightarrow \alpha\langle next^+\rangle\beta \wedge \forall\gamma : \alpha\langle next^+\rangle\gamma \rightarrow \beta\langle next^*\rangle\gamma \tag{4}$$

This inversion shows that *next* can be expressed using AF^R formulas. However, caution must be practiced when using the elimination above, because it may introduce alternations (see [2]). Nevertheless our experiments demonstrate that in a number of commonly occurring examples, the alternation can be removed or otherwise avoided, yielding an equivalent AF^R formula.

2.3 Generating AE^R Verification Conditions

Given a program annotated with loop invariants and procedure specifications, it is possible to automatically generate VCs to check that the invariants are satisfied by all program executions (e.g., see [8]). For example, the VC of $reverse$ asserts that every execution which starts in a state satisfying I_0 satisfies I_3 and that I_3 is indeed *inductive*. That is, if it holds on the loop entry and if the loop is executed, I_3 remains true after the execution. Finally, the VC asserts that I_3 and the negation of the loop condition implies the postcondition I_9.

For simplicity, we do not handle deallocation operations here. Since our logic expresses reachability it does not depend on a particular memory abstraction, and can handle both garbage collection and programs with explicit deallocation.

Unfortunately showing validity of formulas with transitive closure and quantifier alternations, i.e., nesting existential inside universal quantifiers or vice versa is very difficult for first-order theorem provers: existing decision procedures cannot handle such formulas, because even the simplest use of transitive closure leads to undecidability [11].

In this paper we show that for programs with AF^R assertions manipulating singly- and doubly-linked lists, the generated VCs are effectively propositional. However, AF^R formulas are not powerful enough to describe the VCs of programs with AF^R invariants. The main reason is that the semantics of accessing heap fields, e.g., $x := y.next$ requires one level of alternation. Therefore, we slightly generalize AF^R and generate VCs that have the form $\forall^*\exists^* : \varphi$ where φ is a quantifier-free formula which does not contain function symbols in terms but may contain reachability and relation symbols. Validity of formulas in this class, AE^R, are decidable since their negations have the

form $\exists^*\forall^*{:}\varphi$, that is, they belong to the Bernays-Schönfinkel class of formulas [19]. In fact, the formulas can be checked with a SAT solver by replacing existential quantifiers with distinct Skolem constants, and then grounding all universally quantified variables by all combinations of constants. Indeed, Z3 handles these formulas in a precise manner without the need to perform this transformation.

We show that AE^R formulas are closed under weakest preconditions (*wp*), i.e., for every statement S and postcondition Q expressed as AE^R formula, it must be the case that $wp(S, Q)$ is expressed as an AE^R formula. To show this closure property of AE^R formulas, we rely on recent results in descriptive complexity which prove that for singly-linked data structures edge mutations are expressible *without* quantifications [10]. Specifically, this means that updates to the reachability relation, wrt. pointer removals and additions, can be expressed using quantifier-free formulas. We note, however, that our applications to program verification go beyond descriptive complexity in several major ways: (i) Programs can create fresh nodes as a result of dynamic allocation statements of the form $x := \mathsf{new}$. (ii) A heap field read, $x := y.next$, does not mutate the heap but can affect the truth value of reachability constraints. (iii) Calls to libraries can mutate the heap in an unbounded way. (iv) In order to guarantee correctness of loops and procedures, the verification is conducted modularly using AF^R invariants, pre- and postconditions. For example, to verify the correctness of a code which includes a procedure call, we assert that the states at the call satisfy the procedure's precondition expressed as an AF^R formula and assert that after the call the state satisfies the procedure's postcondition specified by an AF^R formula.

Handling Destructive Updates. We first handle the case of statements that assign null to pointer fields and so remove directed paths. For example, statement 5 in the *reverse* program is modeled by

$$wp(c.next := \mathsf{null}, Q) \stackrel{\text{def}}{=} \begin{array}{l} c \neq null \\ \wedge\ Q[\alpha\langle next^*\rangle\beta \wedge (\neg\alpha\langle next^*\rangle c \vee \beta\langle next^*\rangle c)/\alpha\langle next^*\rangle\beta] \end{array} \tag{5}$$

The assignment removes the outgoing edge from the node pointed to by c. This is a simplified condition that also uses the fact that the manipulated list is acyclic. An operation of the form $c.next := \mathsf{null}$ deletes an existing path between nodes α and β if the path goes through a (non-null) node c. This situation can be expressed by the formula $\alpha\langle next^*\rangle c \wedge \neg\beta\langle next^*\rangle c$. So the negation of this formula conjoined with $\alpha\langle next^*\rangle\beta$ must hold in the precondition so that $\alpha\langle next^*\rangle\beta$ holds in the postcondition. Notice that this rule drastically differs from the standard McCarthy axiom [16], which directly assigns a new value to the heap:

$$wp'(c.next := \mathsf{null}, Q) \stackrel{\text{def}}{=} Q[next[c \mapsto null]/next]$$

We forbid the use of this rule for it uses a function (*next*) and relies on "recomputing" reachability constraints in Q by using the transitive closure of $next[c \mapsto null]$. Instead, we directly update the effect on the reachability relation $\alpha\langle next^*\rangle\beta$ by substituting it with a quantifier-free formula shown in (5). A similar definition exists for *wp* for statements like $c.next := d$ that add edges, as we show later in Table 3.

Surprisingly, the semantics of field dereference statement $t := c.next$ is a bit more subtle despite the fact that such a statement does not modify the heap. However, a *wp* for field dereference can also be given in AE^R (see Section 3), thus enabling verification with a SAT solver in a complete way.

As shown by Hesse [10], a QF^R definition of the effect on reachability can be also done for cyclic data structures with a single pointer field. However, for programs with reachability over more than one field in general DAGs, quantifiers are required [6].

2.4 Decidability of AE^R

Reachability constraints written as $\alpha\langle next^*\rangle\beta$ are not directly expressible in FOL. However, AE^R formulas can be reduced to first-order $\forall^*\exists^*$ formulas without function symbols (which are decidable; see Section 2.3) in the following fashion: Introduce a new binary relation symbol $\widehat{n^*}$ with the intended meaning that $\widehat{n^*}(\alpha,\beta) \Leftrightarrow \alpha\langle next^*\rangle\beta$. Even though $\widehat{n^*}$ is an uninterpreted relation, we will consistently maintain the fact that it models reachability. Every formula φ is translated into

$$\varphi' \stackrel{\text{def}}{=} \varphi[\widehat{n^*}(t_1,t_2)/t_1\langle next^*\rangle t_2]$$

For example, the acyclicity relation shown in (1) is translated into:

$$\widehat{ac} \stackrel{\text{def}}{=} \forall\alpha,\beta : \widehat{n^*}(\alpha,\beta) \wedge \widehat{n^*}(\beta,\alpha) \rightarrow \alpha = \beta \tag{6}$$

We add the consistency rule Γ_{linOrd} shown in Table 2, which requires that $\widehat{n^*}$ is a total order. In Section 3 and in [2] we prove that the translated formula $\Gamma_{\text{linOrd}} \rightarrow \varphi'$ is valid if and only if the original formula φ is valid. The proof constructs real models from "simulated" FO models using the reachability inversion (4).

Table 2. Γ_{linOrd} says all points reachable from a given point are linearly ordered

$$\begin{array}{|l|l|}\hline \Gamma_{\text{linOrd}} \stackrel{\text{def}}{=} & \forall\alpha,\beta : \widehat{n^*}(\alpha,\beta) \wedge \widehat{n^*}(\beta,\alpha) \leftrightarrow \alpha = \beta \quad \wedge \\ & \forall\alpha,\beta,\gamma : \widehat{n^*}(\alpha,\beta) \wedge \widehat{n^*}(\beta,\gamma) \rightarrow \widehat{n^*}(\alpha,\gamma) \quad \wedge \\ & \forall\alpha,\beta,\gamma : \widehat{n^*}(\alpha,\beta) \wedge \widehat{n^*}(\alpha,\gamma) \rightarrow (\widehat{n^*}(\beta,\gamma) \vee \widehat{n^*}(\gamma,\beta)) \\ \hline \end{array}$$

2.5 Expressivity of AF^R

Although AF^R is a relatively weak logic, it can express interesting properties of lists. Typical predicates that express disjointness of two lists and sharing of tails are expressible in AF^R. For example, for two singly-linked lists with headers h, k, $disjoint(h,k) \Leftrightarrow \forall\alpha : \alpha \neq null \rightarrow \neg(h\langle next^*\rangle\alpha \wedge k\langle next^*\rangle\alpha)$.

Another capability still within the power of AF^R is to relax the earlier assumption that the program manipulates the whole memory. We describe a summary of $reverse$ on arbitrary acyclic linked lists in a heap that may contain other linked data structures. Realistic programs obey ownership requirements, e.g., the head h of the list *owns* the input list which means that it is impossible to reach one of the list nodes without passing through h. That is,

$$\forall\alpha,\beta : \alpha \neq null \rightarrow (h\langle next^*\rangle\alpha \wedge \beta\langle next^*\rangle\alpha) \rightarrow h\langle next^*\rangle\beta \tag{7}$$

This requirement is conjoined to the precondition, ac, of $reverse$. Its postcondition is the conjunction of ac, the fact that h_0 and d reach the same nodes, (i.e., $\forall\alpha : h_0\langle next_0^*\rangle\alpha \Leftrightarrow d\langle next^*\rangle\alpha$) and

$$\forall\alpha,\beta : \alpha\langle next^*\rangle\beta \Leftrightarrow \left\{\begin{array}{ll} \beta\langle next_0^*\rangle\alpha \wedge \beta \neq null & h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \\ \alpha\langle next_0^*\rangle\beta & \neg h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \\ \mathbf{false} & h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \\ \alpha\langle next_0^*\rangle h_0 \wedge \beta = h_0 & \neg h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \end{array}\right\} \quad (8)$$

Here, the bracketed formula should be read as a four-way case, i.e., as disjunction of the formulas $h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \wedge \beta\langle next_0^*\rangle\alpha \wedge \beta \neq null$; $\neg h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \wedge \alpha\langle next_0^*\rangle\beta$; $h_0\langle next_0^*\rangle\alpha \wedge \neg h_0\langle next_0^*\rangle\beta \wedge \mathbf{false}$; and, $\neg h_0\langle next_0^*\rangle\alpha \wedge h_0\langle next_0^*\rangle\beta \wedge \alpha\langle next_0^*\rangle h_0 \wedge \beta = h_0$. Intuitively, this summary distinguishes between the following four cases: (i) both the source (α) and the target (β) are in the reversed list (ii) both source and target are outside of the reversed list (iii) the source is in the reversed list and the target is not, and (iv) the source is outside and the target is in the reversed list. Cases (i)–(iii) are self-explanatory. For (iv) reachability can occur when there exists a path from α to $h_0 = \beta$. Formula (8) is in AF^R. In terms of [21], this means that we assume that the procedure is cutpoint free. We can also generate an AF^R summary for a program with *fixed* number of cutpoints, as is done in Section 5.

The general case of unbounded number of cutpoints requires a formula that is outside AF^R. A non-AF^R formula also arises when we want to express that a program manipulates two lists of equal length; such a formula requires an inductive definition. See [2] for examples of these formulas.

3 Weakest Preconditions of Atomic Heap Manipulating Statements

In this section we show how to express the weakest liberal preconditions of atomic heap manipulating statements using AE^R formulas, for programs that manipulate acyclic singly-linked lists. Table 3 shows standard *wp* computation rules (top part) and the corresponding rules for field update, field read and dynamic allocation (bottom part). The correctness of the rule for destructive field update is according to Hesse's thesis [10].

Field Dereference. The rationale behind the formula for $wp(x := y.next, Q)$ is that if y has a successor, then the formula Q should be satisfied when x is replaced by this successor. The natural way to specify this is using the Hoare assignment rule

$$wp'(x := y.next, Q) \stackrel{\text{def}}{=} Q[next(y)/x]$$

However, this rule uses the function *next* and does not directly express reachability. Instead we will construct a relation r_{next} such that $r_{next}(\alpha,\beta) \Leftrightarrow next(\alpha) = \beta$ and then use universal quantifications to "access" the value

$$wp''(x := y.next, Q) \stackrel{\text{def}}{=} \forall\alpha : r_{next}(y,\alpha) \rightarrow Q[\alpha/x]$$

Table 3. Rules for computing weakest liberal preconditions for procedures annotated with loop invariants and postconditions. I denotes the loop invariant, $[\![B]\!]$ is the AF^R formula for program conditions and Q is the postcondition expressed as an AF^R formula. The top frame shows the standard *wp* rules for While-language, the bottom frame contains our additions for heap updates, memory allocation, and dereference.

$$
\begin{aligned}
wp(\mathsf{skip}, Q) &\stackrel{\text{def}}{=} Q \\
wp(x := y, Q) &\stackrel{\text{def}}{=} Q[y/x] \\
wp(S_1 \ ; \ S_2, Q) &\stackrel{\text{def}}{=} wp(S_1, wp(S_2, Q)) \\
wp(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, Q) &\stackrel{\text{def}}{=} [\![B]\!] \wedge wp(S_1, Q) \vee \\
&\qquad \neg[\![B]\!] \wedge wp(S_2, Q) \\
wp(\mathsf{while}\ B\ \{I\}\ \mathsf{do}\ S, Q) &\stackrel{\text{def}}{=} I
\end{aligned}
$$

$$
\begin{aligned}
wp(x.next := \mathsf{null}, Q) &\stackrel{\text{def}}{=} Q[\alpha\langle next^*\rangle\beta \wedge (\neg\alpha\langle next^*\rangle x \vee \beta\langle next^*\rangle x)/\alpha\langle next^*\rangle\beta] \\
wp(x.next := y, Q) &\stackrel{\text{def}}{=} \neg y\langle next^*\rangle x \wedge \\
&\qquad Q[\alpha\langle next^*\rangle\beta \vee (\alpha\langle next^*\rangle x \wedge y\langle next^*\rangle\beta)/\alpha\langle next^*\rangle\beta] \\
wp(x := \mathsf{new}, Q) &\stackrel{\text{def}}{=} \forall\alpha : \left(\textstyle\bigwedge_{p \in Pvar \cup \{null\}} \neg p\langle next^*\rangle\alpha\right) \rightarrow Q[\alpha/x] \\
P_{next^+} &\stackrel{\text{def}}{=} s\langle next^*\rangle t \wedge s \neq t \\
P_{next} &\stackrel{\text{def}}{=} P_{next^+} \wedge \forall\gamma : P_{next^+}[\gamma/t] \rightarrow \gamma\langle next^*\rangle t \\
wp(x := y.next, Q) &\stackrel{\text{def}}{=} \forall\alpha : P_{next}[y/s, \alpha/t] \rightarrow Q[\alpha/x]
\end{aligned}
$$

Since *next* is acyclic, we can express r_{next} in terms of *next** as follows. First we observe that $next(\alpha) \neq \alpha$. Also, since *next* is a function, the set of nodes reachable from α is totally ordered by *next**. Therefore, similarly to Section 2.2, we can express $r_{next}(\alpha, \beta)$ as the minimal node β in this order where $\beta \neq \alpha$. Expressing minimality "costs" one extra universal quantification.

In Table 3, formula P_{next} expresses r_{next} in terms of *next**: P_{next} holds if and only if there is a path of length 1 between s and t (source and target). Thus, $P_{next}[y/s, \alpha/t]$ is satisfied exactly when $\alpha = next(y)$. If y does not have a successor, then $P_{next}[y/s, \alpha/t]$ can only be $\mathbf{true}$ if $\alpha = null$, hence Q should be satisfied when x is replaced by *null*, which is in line with the concrete semantics. A central lemma in [2] shows that the formula P_{next} correctly defines *next* as a relation.

Dynamic Allocation. The rule $wp(x := \mathsf{new}, Q)$ expresses the semantic uncertainty caused by the behavior of the memory allocator. We want to be compatible with any run-time memory management, so we do not enforce a concrete allocation policy, but require that the allocated node meets some reasonable specifications, namely, that it is different from all values stored in program variables, and that it is unreachable from any other node allocated previously (Note: for programs with explicit `free()`, this assumption relies on the absence of dangling pointers, which can be verified by introducing appropriate assertions; this is, however, beyond the scope of this paper).

4 Generating an AE^R Verification Condition

Table 4 provides the standard rules for computing VCs using weakest liberal preconditions. An auxiliary function VC_{aux} is used for defining the set of side conditions for the loops occurring in the program. These rules are standard and their soundness and relative completeness have been discussed elsewhere (e.g. see [8]).

We assume that the effect, $[\![B]\!]$, of the condition B used in the conditional and the while loop, is defined by an AF^R formula. We also assume that all loop invariants I, the precondition P, and postcondition Q are AF^R formulas. The rule for while loop is split into two parts: in the *wp* we take just the loop invariant, where VC_{aux} asserts that loop invariants are inductive and implies the postcondition for each loop.

The rules may generate exponential formulas. Another solution can be implemented either using the method of Flanagan and Saxe [7] or by using a set of symbols for every program point.

Table 4. Standard rules for computing VCs using weakest liberal preconditions for procedures annotated with loop invariants and pre/postconditions

$VC_{aux}(S, Q) \stackrel{\text{def}}{=} \varnothing$ (for any atomic command S)
$VC_{aux}(S_1; S_2, Q) \stackrel{\text{def}}{=} VC_{aux}(S_1, wp(S_2, Q)) \cup VC_{aux}(S_2, Q)$
$VC_{aux}(\mathsf{if}\ B\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2, Q) \stackrel{\text{def}}{=} VC_{aux}(S_1, Q) \cup VC_{aux}(S_2, Q)$
$VC_{aux}(\mathsf{while}\ B\ \{I\}\ \mathsf{do}\ S, Q) \stackrel{\text{def}}{=} VC_{aux}(S, I)\ \cup$ $\{I \wedge [\![B]\!] \rightarrow wp(S, I), I \wedge \neg[\![B]\!] \rightarrow Q\}$
$VC_{gen}(\{P\}S\{Q\}) \stackrel{\text{def}}{=} P \rightarrow wp(S, Q) \wedge \bigwedge VC_{aux}(S, Q)$

Notice that Table 4 only uses weakest liberal preconditions in a positive context without negations. Therefore, the following proposition (proof in [2]) holds.

Proposition 1 (VCs in AE^R). *For every program S whose precondition P, postcondition Q, branch conditions, loop conditions, and loop invariants are all expressed as AF^R formulas, $VC_{gen}(\{P\}S\{Q\})$ is in AE^R.*

Optimization Remark. The size of the VC can be significantly reduced if instead of syntactic substitution, we introduce a new vocabulary for each substituted atomic formula, axiomatizing its meaning as a separate formula. For example, $Q[P(\alpha, \beta)/\alpha\langle next^*\rangle\beta]$ (where P is some formula with free variables α, β), can be written more compactly as $Q[r_1(\alpha, \beta)/\alpha\langle next^*\rangle\beta] \wedge \forall\alpha, \beta : r_1(\alpha, \beta) \Leftrightarrow P(\alpha, \beta)$, where r_1 is a fresh relational symbol. When Q contains many applications of $\langle next^*\rangle$ and P is large, this may save a lot of formula space; roughly, it reduces the order of the VC size from quadratic to linear. Our original implementation employed this optimization, which is also nice for finding bugs — when the program violates the invariants the SAT solver produces a counterexample with the concrete states at every program point. The approach of [7] is also applicable in this case.

5 Extensions

Doubly-linked List and Nested Lists. To verify a program that manipulates a doubly-linked list, all that needs to be done is to duplicate the analysis we did for *next*, for a second pointer field *prev*. As long as the only atomic formulas used in assertions are $\alpha\langle \mathit{next}^* \rangle\beta$ and $\alpha\langle \mathit{prev}^* \rangle\beta$ (and not, for example, $\alpha\langle(\mathit{next}|\mathit{prev})^*\rangle\beta$), providing the substitutions for atomic formulas in Table 3 would not get us outside of the class AE^R. In particular, we have verified the doubly-linked list property:

$$\forall\alpha,\beta : h\langle \mathit{next}^* \rangle\alpha \wedge h\langle \mathit{next}^* \rangle\beta \rightarrow (\alpha\langle \mathit{next}^* \rangle\beta \Leftrightarrow \beta\langle \mathit{prev}^* \rangle\alpha).$$

In fact we can verify nested lists and, in general, lists with arbitrary number of pointer fields as long as reachability constraints are expressed using only one function symbol at a time, like in the case of $next$ and $prev$ above.

Cycles. For data structures with a single pointer, the acyclicity restriction may be lifted by using an alternative formulation that keeps and maintains more auxiliary information [10,13]. Instead of keeping track of just *next**, we instrument the edge addition operation with a check: if the added edge is about to close a cycle, then instead of adding the edge, we keep it in a separate set M of "cycle-inducing" edges. Two properties of lists now come into play: (1) The number of cycles reachable from program variables, and hence the size of M, is bounded by the number of program variables; (2) Any path (simple or otherwise) in the heap may utilize at most one of those edges, because once a path enters a cycle, there is no way out. In all assertions, therefore, we replace $\alpha\langle \mathit{next}^* \rangle\beta$ with: $\alpha\langle \mathit{next}^* \rangle\beta \vee \bigvee_{\langle u,v\rangle\in M}(\alpha\langle \mathit{next}^* \rangle u \wedge v\langle \mathit{next}^* \rangle\beta)$. Notice that it is possible to construct this formula thanks to the bound on the size of M; otherwise, an existential quantifier would have been required in place of the disjunction.

More details and cases for cycles can be found in the Technical Report [2].

Bounded Sharing. Arbitrary sharing in data structures is hard, because even in lists, any node of the list may be shared (that is, have more than one incoming edge). In this case we have to use quantification since we do not know in advance which node in the list is going to be a cutpoint for which other nodes. However, when the *entire* heap consists solely of lists, the quantifier may be replaced with a disjunction if we take into account that there is a bounded number of program variables, which can serve as the heads of lists, and any two lists have at most one cutpoint. Such heaps when viewed as graphs are much simpler than general DAGs, since one can define in advance a set of *constant symbols* to hold the edges that induce the sharing; for example, if we have one list through the nodes $x \rightarrow u_1 \rightarrow u_2$ and a second list through $y \rightarrow v_1 \rightarrow v_2$, all distinct locations, then adding an edge $u_2 \rightarrow v_1$ would create sharing, as the nodes v_1, v_2 become accessible from both x and y. This technique is also covered by Hesse [10].

6 Composing Procedure Summaries to Check Program Equivalence

This section argues that AF^R-postconditions of procedure summaries can be sequentially composed and used to check if two pieces of code are equivalent, i.e., that they produce the same output for a given input.

Illustrating $reverse(reverse\ h) = h$. Let next_1^* denote the reachability after running the inner $reverse$, and let next_2^* denote the reachability after running the outer $reverse$. We can express the equivalence of $reverse(reverse\ h)$ and h using the following AF^R implication:

$$\begin{aligned}(\forall \alpha, \beta : \alpha\langle \mathit{next}_1^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_0^*\rangle\alpha) \wedge (\forall \alpha, \beta : \alpha\langle \mathit{next}_2^*\rangle\beta \Leftrightarrow \beta\langle \mathit{next}_1^*\rangle\alpha) \\ \rightarrow \forall \alpha, \beta : \alpha\langle \mathit{next}_2^*\rangle\beta \Leftrightarrow \alpha\langle \mathit{next}_0^*\rangle\beta\end{aligned} \tag{9}$$

The second conjunct of the implication's antecedent describes the effect of the inner $reverse$ on the initial state while the third conjunct describes the effect of the outer $reverse$ on the state resulting from the first. The consequent of the implication states that the initial and final states are equivalent.

Illustrating $\mathit{filter}(C, reverse(h)) = reverse(\mathit{filter}(C, h))$. The program *filter* takes a unary predicate C on nodes, and a list with head h, and returns a list with all nodes satisfying C removed. The postcondition of *filter* is: $\forall \alpha, \beta : \alpha\langle \mathit{next}^*\rangle\beta \Leftrightarrow \neg C(\alpha) \wedge \neg C(\beta) \wedge \alpha\langle \mathit{next}_0^*\rangle\beta$. It says that β is reachable from α in the filtered list provided neither α nor β satisfies C and β was reachable from α initially. We show ([2]) that the equivalence of $\mathit{filter}(C, reverse(h))$ and $reverse(\mathit{filter}(C, h))$ can be expressed using an AF^R implication.

7 Experimental Results

7.1 Details

We have implemented a VC generator, according to Tables 3 and 4, in Python, and PLY (Python Lex-Yacc) is employed at the front-end to parse While-language programs annotated with AF^R assertions. The tool verifies that invariants are in the class AF^R and have reachability constraints along a single field (of the form f^*). The assertions may refer to the store and heap at the entry to the procedure via x_0, f_0, etc. SMT-LIB v2 [1] standard notation is used to format the VC and to invoke Z3. The validity of the VC can be checked by providing its negation to Z3. If Z3 exhibits a satisfying assignment then that serves as counterexample for the correctness of the assertions. If no satisfying assignment exists, then the generated VC is valid, and therefore the program satisfies the assertions.

The output model/counterexample (S-Expression), if one is generated, is then also parsed, so that we have the truth table of next^*. This structure represents the state of the program either at entry or at the beginning of a loop iteration: running the program from this point will violate one or more invariants. To provide feedback to the user, *next* is recovered by computing (4), and then the `pygraphviz` tool is used to visualize and present to the user a directed graph, whose vertices are nodes in the heap, and whose edges are the *next* pointer fields.

We also implemented two procedures for generating VCs: the first one implements the standard rules shown in Table 4 and a second one uses a separate set of relation and constant symbols per program point as a way to reduce the size of the generated VC formula. We only report data on the former since it exhibited better running times.

7.2 Verification Examples

We have written AF^R loop invariants and procedure pre- and postconditions for 13 example procedures shown in Table 6. These are standard benchmarks and what they do can be inferred either from their names or from Table 5. We are encouraged by the fact that it was not difficult to express assertions in AF^R for these procedures. The annotated examples and the VC generation tool are publicly available from `http://www.cs.tau.ac.il/~shachar/afwp.html`.

For an example of the annotations used in the benchmarks, see Table 1, listing the precondition, loop invariant, and postcondition of $reverse$.

As expected, Z3 is able to verify all the correct programs. Table 6 shows statistics for size and complexity of the invariants and the running times for Z3.

To give some account of the programs' sizes, we observe the program summary specification given as pre- and postcondition, count the number of atomic formulas in each of them, and note the depth of quantifier nesting; all our samples had only universal quantifiers. We did the same for each program's loop invariant and for the generated VC_{gen}. Naturally, the size of the VC grows rapidly —approximately at a quadratic rate. This can be observed in the result of the measurements for "SLL: merge", where (i) the size of the invariant and (ii) the number of if-branches and heap manipulating statements, was larger than those in other examples. Still, the time required by Z3 to prove that the VC is valid is short.

For comparison, the size of the formula generated by the alternative implementation, using a separate set of symbols for each program location, was about 10 times shorter — 239 atomic formulas. However, Z3 took a significantly longer time, at 1357ms. We therefore preferred to use the first implementation.

Thanks to the fact that FOL-based tools, and in particular SAT solvers, permit multiple relation symbols we were able to express ordering properties in sorted lists, and so verify order-aware programs such as "insert" and "merge". This situation can be contrasted with tools like Mona ([12],[9]) which are based on monadic second-order logic, where multiple relation symbols are disallowed.

Additionally, we made experiments in composing summaries of $filter$ and $reverse$ (Section 6). In this case, we wrote the formulas manually and ran Z3 on them, to get a proof of the validity of the equivalences.

Table 5. Description of some linked list manipulating programs verified by our tool

SLL: insert	— Adds a node into a sorted list, preserving order.
SLL: find	— Locates the first item in the list with a given value.
SLL: last	— Returns the last node of the list.
SLL: merge	— Merges two sorted lists into one, preserving order.
SLL: swap	— Exchanges the first and second element of a list.
DLL: fix	— Directs the back-pointer of each node towards the previous node, as required by data structure invariants.
DLL: splice	— Splits a list into two well-formed doubly-linked lists.

Benchmark	Formula size P,Q # ∀	I # ∀	VC # ∀	Solving time (Z3)
SLL: reverse	2 2	11 2	133 3	57ms
SLL: filter	5 1	14 1	280 4	39ms
SLL: create	1 0	1 0	36 3	13ms
SLL: delete	5 0	12 1	152 3	23ms
SLL: deleteAll	3 2	7 2	106 3	32ms
SLL: insert	8 1	6 1	178 3	17ms
SLL: find	7 1	7 1	64 3	15ms
SLL: last	3 0	5 0	74 3	15ms
SLL: merge	14 2	31 2	2255 3	226ms
SLL: rotate	6 1	- -	73 3	22ms
SLL: swap	14 2	- -	965 5	26ms
DLL: fix	5 2	11 2	121 3	32ms
DLL: splice	10 2	- -	167 4	27ms

Table 6. Implementation Benchmarks; P,Q — program's specification given as pre- and post-condition, I — loop invariant, VC — verification condition, # — number of atomic formulas, ∀ — quantifier nesting

The tests were conducted on a 1.7GHz Intel Core i5 machine with 4GB of RAM, running OS X 10.7.5. The version of Z3 used was 4.2, compiled for 64-bit Intel architecture (using `gcc` 4.2, LLVM). The solving time reported is wall clock time of the execution of Z3.

7.3 Buggy Examples

We also applied the tool to erroneous programs and programs with incorrect assertions. The results, including run-time statistics and formula sizes, are reported in Table 7. In addition, we measured the size of the model generated, by observing the size of the generated domain—which reflects the number of nodes in the heap. As expected, Z3 was able to produce concrete counterexample of a small size. Since these are slight variations of the correct programs, size and running time statistics are similar.

An example of generated output when a program fails to verify can be seen, for the $insert$ program, in Fig. 3. The tool reports, as part of its output, that counterexample occurs when $j = \mathit{null}$ and $h.val = i.val = e.val$.

```
Node insert(Node h, Node e) {
  Node i = h, j = null;
  while (i != null && e.val >= i.val) {
    j = i; i = i.n;
  }
  if (j != null) { j.n = e; e.n = i; }
  else { e.n = h; h = e; }
  return h;
}
```

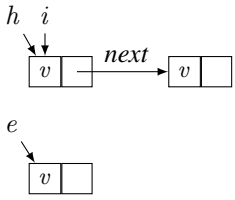


Fig. 3. Sample counterexample generated for a buggy version of $insert$. Here, the loop invariant required that $\forall\alpha : (h\langle \mathit{next}^*\rangle\alpha \wedge \neg i\langle \mathit{next}^*\rangle\alpha) \rightarrow \alpha <_{val} e$ (where $<_{val}$ is an ordering on nodes according to their values), but the loop will execute one more time, violating this.

Table 7. Information about benchmarks that demonstrate detection of several kinds of bugs in pointer programs. In addition to the previous measurements, the last column lists the size of the generated counterexample in terms of the number of vertices, or linked-list nodes.

Benchmark	Nature of Defect	Formula size P,Q #	P,Q ∀	I #	I ∀	VC #	VC ∀	Solving time (Z3)	C.e. size (|L|)
SLL: find	*null* pointer dereference.	7	1	7	1	64	3	18ms	2
SLL: deleteAll	Loop invariant in annotation is too weak to prove the desired property.	3	2	5	2	68	3	58ms	5
SLL: rotate	Transient cycle introduced during execution.	6	1	-	-	109	3	25ms	3
SLL: insert	Unhandled corner case when an element with the same value already exists in the list — ordering violated.	8	1	6	1	178	3	33ms	4

8 Discussion

8.1 Related Work

Decidable Logic. The results in this paper show that reachability properties of programs manipulating linked lists can be verified using a simple decidable logic AE^R. Many recent decidable logics for reasoning about linked lists have been proposed [17,22,15,3]. In comparison to these works we drastically restrict the way quantifiers are allowed but permit arbitrary use of relations. Thus, strictly speaking our logic is incomparable to the above logics. We show that relations are used even in programs like $reverse$ to write procedure summaries such as the one in (8) and for expressing numeric orders in sorting programs.

Employing Theorem Provers. The seminal paper on program verification [18] provides useful axioms for verifying reachability in linked data structures using theorem provers and conjectures that these axioms are complete for describing reachability. Lev-Ami et al. [14] show that no such axiomatization is possible. The current submission sidesteps the above impossibility results by restricting first order quantifications and by using the fact that Bernays-Schönfinkel formulas have finite model property.

Lahiri and Qadeer [13] provide rules for weakest of preconditions for programs with circular linked lists. The formulas are similar to Hesse's [10] but require that the programmer explicitly break the cycle. Our framework can be used both with and without the help of the programmer. In practice it may be beneficial to require that the programmer breaks the cycle in certain cases in order to allow invariants which distinguish between segments in the cycle.

Descriptive Complexity. Descriptive complexity was recently incorporated into the TVLA shape analysis framework [20]. In this paper we pioneer the use of descriptive complexity for *guaranteeing* that if the programmer writes AF^R assertions and if the program manipulates singly- and doubly-linked lists, then the VCs are guaranteed to be expressible as AE^R formulas.

8.2 Conclusion

The results in this paper shed some light on the complexity of reasoning about programs that manipulate linked data structures such as singly- and doubly-linked lists. The invariants in many of these programs can be expressed without quantifier alternation. Alternations are introduced by unbounded cutpoints and reasoning about more complicated directed acyclic graphs. Furthermore, for programs manipulating general graphs higher order reasoning may be required.

References

1. SMTLIB: Satisfiability modulo theories library, `http://smtlib.cs.uiowa.edu/docs.html`
2. Technical report, `http://www.cs.tau.ac.il/~shachar/dl/tr-2013.pdf`
3. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012)
4. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
5. Demetrescu, C., Italiano, G.F.: Decremental all-pairs shortest paths. In: Encyclopedia of Algorithms (2008)
6. Dong, G., Su, J.: Incremental maintenance of recursive views using relational calculus/sql. SIGMOD Record 29, 44–51 (2000)
7. Flanagan, C., Saxe, J.B.: Avoiding exponential explosion: generating compact verification conditions. In: POPL (2001)
8. Frade, M., Pinto, J.: Verification conditions for source-level imperative programs. Computer Science Review 5(3), 252–277 (2011)
9. Henriksen, J., Jensen, J., Jørgensen, M., Klarlund, N., Paige, B., Rauhe, T., Sandholm, A.: Mona: Monadic second-order logic in practice. In: Brinksma, E., Steffen, B., Cleaveland, W.R., Larsen, K.G., Margaria, T. (eds.) TACAS 1995. LNCS, vol. 1019, pp. 89–110. Springer, Heidelberg (1995)
10. Hesse, W.: Dynamic computational complexity. PhD thesis, Dept. of Computer Science, University of Massachusetts, Amherst, MA (2003)
11. Immerman, N., Rabinovich, A., Reps, T., Sagiv, M., Yorsh, G.: The boundary between decidability and undecidability for transitive-closure logics. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 160–174. Springer, Heidelberg (2004)
12. Kautz, H., Selman, B.: Knowledge compilation and theory approximation. J. ACM 43(2), 193–224 (1996)
13. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using smt solvers. In: POPL (2008)
14. Lev-Ami, T., Immerman, N., Reps, T.W., Sagiv, M., Srivastava, S., Yorsh, G.: Simulating reachability using first-order logic with applications to verification of linked data structures. Logical Methods in Computer Science 5(2) (2009)

15. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL (2011)
16. McCarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28 (1962)
17. Møller, A., Schwartzbach, M.I.: The pointer assertion logic engine. In: PLDI (2001)
18. Nelson, G.: Verifying reachability invariants of linked structures. In: POPL (1983)
19. Piskac, R., de Moura, L.M., Bjørner, N.: Deciding effectively propositional logic using dpll and substitution sets. J. Autom. Reasoning 44(4), 401–424 (2010)
20. Reps, T.W., Sagiv, M., Loginov, A.: Finite differencing of logical formulas for static analysis. ACM Trans. Program. Lang. Syst. 32(6) (2010)
21. Rinetzky, N., Bauer, J., Reps, T.W., Sagiv, S., Wilhelm, R.: A semantics for procedure local heaps and its abstractions. In: POPL (2005)
22. Yorsh, G., Rabinovich, A.M., Sagiv, M., Meyer, A., Bouajjani, A.: A logic of reachable patterns in linked data-structures. J. Log. Algebr. Program 73(1-2), 111–142 (2007)

Automating Separation Logic Using SMT

Ruzica Piskac[1], Thomas Wies[2], and Damien Zufferey[3]

[1] MPI-SWS, Germany
[2] New York University, USA
[3] IST Austria

Abstract. Separation logic (SL) has gained widespread popularity because of its ability to succinctly express complex invariants of a program's heap configurations. Several specialized provers have been developed for decidable SL fragments. However, these provers cannot be easily extended or combined with solvers for other theories that are important in program verification, e.g., linear arithmetic. In this paper, we present a reduction of decidable SL fragments to a decidable first-order theory that fits well into the satisfiability modulo theories (SMT) framework. We show how to use this reduction to automate satisfiability, entailment, frame inference, and abduction problems for separation logic using SMT solvers. Our approach provides a simple method of integrating separation logic into existing verification tools that provide SMT backends, and an elegant way of combining SL fragments with other decidable first-order theories. We implemented this approach in a verification tool and applied it to heap-manipulating programs whose verification involves reasoning in theory combinations.[1]

1 Introduction

Separation logic (SL) [24] is an extension of Hoare logic for proving the correctness of heap-manipulating programs. Its great asset lies in its assertion language, which can succinctly express how data structures are laid out in memory. This language has two characteristic features: it provides 1) a *spatial conjunction* operator that decomposes the heap into disjoint regions, each of which can be reasoned about independently (this enables an elegant treatment of pointer aliasing); and 2) *inductive spatial predicates* that describe the shape of unbounded linked data structures such as lists, trees, etc. SL assertions give rise to the so-called *frame rule*, a Hoare-logic proof rule that enables compositional verification of heap-manipulating programs.

The frame rule makes separation logic attractive for developers of program verification tools [6, 8, 19, 20, 35]. However, the logic also poses a challenge to automation: it is a non-classical logic that requires specialized symbolic execution engines for encoding the behavior of programs, and specialized theorem provers for discharging the generated proof obligations. Existing SL-based tools therefore implement their own tailor-made theorem provers. This brings its own challenges.

First, extending existing verification tools that rely on specifications written in classical first-order logic with SL support is a significant effort. The tailor-made SL provers cannot be easily integrated with the theorem provers used by such tools. Second, the

[1] An extended version of this paper is available as a technical report [28].

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 773–789, 2013.

analysis of real-world programs involves more than just reasoning about heap structures. For instance, the combination of linked data structures and pointer arithmetic is pervasive in low-level system code [14, 19]. Other examples include the dynamic reinterpretation of memory (e.g., treating a memory region both as a linked structure and as an array of bit-vectors) and dependencies on data stored in linked structures (e.g., sortedness constraints). To deal with such programs, existing SL tools make simplifying and (deliberately) unsound assumptions about the underlying memory model, rely on interactive help from the user, or implement incomplete extensions to allow some limited support for reasoning about other theories.

The integration of a separation logic prover into an SMT solver can address these challenges. Modern SMT solvers such as CVC4 [3] and Z3 [16] already implement decision procedures for many theories that are relevant in program verification, e.g., linear arithmetic, arrays, and bit-vectors. They also implement generic mechanisms for combining these theories, treating the theory solvers as independent components. These mechanisms provide guarantees about completeness and decidability. A reduction of separation logic to first-order logic enables such complete combinations with other theories. Finally, SMT solvers are already an integral part in the tool chain of many existing verification tools. These tools could directly benefit from an integrated SL prover.

So far, a seamless integration of a separation logic prover and an SMT solver has not yet been realized. This paper represents a step towards achieving this goal.

We propose a technique for SMT-based reasoning about separation logic assertions. Our technique relies on a translation of SL formulas into a decidable fragment of first-order logic, which we refer to as the *logic of graph reachability and stratified sets* (GRASS). Formulas in this logic express properties of the structure of graphs, such as whether nodes in the graph are inter-reachable, as well as properties of sets of nodes. These sets are used to give a natural encoding of the semantics of spatial conjunction, while graph reachability enables reasoning about inductive spatial predicates without relying on induction, which is not well supported by first-order theorem provers.

We show how to use the translation to check satisfiability and entailment of SL formulas. The latter enables automated verification of programs with SL specifications. In particular, it can leverage existing infrastructure for verification condition generation provided by tools such as Boogie [2]. Using a characterization of partial models of GRASS formulas, we further demonstrate how our technique can solve frame inference and abduction problems, which are key to efficient inter-procedural analysis of heap-manipulating programs [11]. Finally, we prove that our translation enables theory combination of separation logic within the Nelson-Oppen combination framework [23].

To demonstrate the feasibility of our approach we have implemented the decision procedure for GRASS using an SMT solver. Based on this implementation we built a prototype tool for inter-procedural analysis of heap-manipulating programs. We successfully used this tool to automatically verify procedures manipulating list-like data structures against specifications expressed in separation logic. Our examples include benchmarks such as sorting algorithms whose verification relies on the combination of heap and arithmetic reasoning.

Related Work. Several decidable fragments of separation logic have been studied in the literature. Most prominent is the fragment of linked lists introduced in [5], which

is now used in extended forms by many SL-based tools. The original paper describes a decision procedure for satisfiability and entailment for the fragment of linked lists. More recently, these problems were shown to be decidable in polynomial time [15] using a graph-based algorithm.

Translations of separation logic into first-order logic have been previously studied in [12] and [9]. The result in [12] does not consider inductive predicates, which are needed for expressing properties of recursive data structures such as lists, trees, etc. The approach in [9] considers inductive predicates but does not target a decidable fragment. Neither of these approaches considers frame inference, abduction, or theory combination. In [26], Perez and Rybalchenko showed that significant performance improvements can be obtained by incorporating first-order theorem proving techniques into SL provers. More recently and concurrently to us, they have also considered the problem of theory combination [27]. The authors of [10] describe another hybrid approach in which an SL decision procedure for entailment is extended to enable reasoning about quantified constraints on data. However, [10] does not address theory combination in general. Also, neither [26] nor [10] consider frame inference or abduction.

Alternatives to separation logic that enable compositional reasoning about heap-manipulating programs but rely on classical logic include (implicit) dynamic frames [21] and region logic [1, 30]. The connection between separation logic and implicit dynamic frames has been studied in [25]. Our reduction of separation logic to first-order logic is in part inspired by region logic, which also uses sets to partition the memory into disjoint regions. A crucial difference from our approach is that region logic has no inbuilt support for recursive data structures.

We build on previous results on SMT-based decision procedures for theories of reachability in graphs [22, 32, 33] and decision procedures for theories of stratified sets [36]. Our logic GRASS combines these two theories and extends them with set comprehensions that define sets of nodes in terms of properties expressed in the logic. This extension is essential to enable a succinct encoding of spatial conjunctions.

2 Preliminaries

We present our approach in many-sorted first-order logic with equality, which is the theoretical foundation of modern SMT solvers. We follow standard notation and conventions for syntax and semantics of first-order logic as defined, e.g., in [29].

Many-Sorted First-Order logic. A *signature* Σ is a tuple (S, Ω, Π), where S is a countable set of *sorts*, Ω is a countable set of *function symbols*, and Π is a countable set of *predicate symbols*. Each function and predicate symbol has an associated sort, which is a tuple of sorts in S. A function symbol whose sort is a single sort in S is called *constant*. For two signatures Σ_1 and Σ_2 we write $\Sigma_1 \cup \Sigma_2$ for the signature that is obtained by taking the point-wise union of Σ_1 and Σ_2. We say that Σ_1 and Σ_2 are *disjoint* if they do not share any function or predicate symbols. We write $\Sigma_1 \subseteq \Sigma_2$ if $\Sigma_1 \cup \Sigma_2 = \Sigma_2$. A Σ-term is built as usual from the function symbols in Ω and variables taken from a set $\mathcal{X}$ that is disjoint from S, Ω, and Π. Each variable $x \in \mathcal{X}$ has an associated sort in S. We also assume the standard notions of Σ-atom, Σ-literal, and Σ-formula.

Interpretations and Structures. In the following, let $\Sigma = (S, \Omega, \Pi)$ be a signature. A *partial Σ-interpretation* $\mathcal{A}$ over variables $\mathcal{X}$ is a function that maps each sort $s \in S$ to a non-empty set $s^{\mathcal{A}}$ and each function symbol $f \in \Omega$ of sort $s_1 \times \cdots \times s_n \to t$ to a partial function $f^{\mathcal{A}} : s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}} \rightharpoonup t^{\mathcal{A}}$. Similarly, every predicate $p \in \Pi$ of sort $s_1 \times \cdots \times s_n$ is interpreted as a relation $p^{\mathcal{A}} \subseteq s_1^{\mathcal{A}} \times \cdots \times s_n^{\mathcal{A}}$. Finally, $\mathcal{A}$ interprets every variable $x \in \mathcal{X}$ of associated sort $s \in S$ by some element $x^{\mathcal{A}} \in s^{\mathcal{A}}$. A partial interpretation $\mathcal{A}$ is called *total interpretation* or simply *interpretation* if it interprets all function symbols by total functions. We denote by $\mathcal{A}|_{\Sigma',\mathcal{X}'}$ the Σ'-interpretation over $\mathcal{X}'$ that is obtained by restricting $\mathcal{A}$ to a signature $\Sigma' \subseteq \Sigma$ and a set of variables $\mathcal{X}' \subseteq \mathcal{X}'$. We further write $\mathcal{A}|_{\Sigma}$ for $\mathcal{A}|_{\Sigma,\emptyset}$. A *(partial) Σ-structure* is a (partial) Σ-interpretation over an empty set of variables. For a partial Σ-interpretation $\mathcal{A}$ and $\sigma \in S \cup \Omega \cup \Pi \cup \mathcal{X}$, we denote by $\mathcal{A}[\sigma \mapsto v]$ the interpretation that is like $\mathcal{A}$ but interprets σ as v.

Given a Σ-interpretation $\mathcal{A}$, the evaluation $t^{\mathcal{A}}$ of a Σ-term t in $\mathcal{A}$ is defined inductively over the structure of t, as usual. Similarly, the evaluation of a Σ-formula in $\mathcal{A}$ is obtained from the interpretation of terms in the usual way. In particular, we use the standard interpretations for equality, propositional connectives, and quantifiers. A quantified variable of sort s ranges over all elements of $s^{\mathcal{A}}$. For a formula F we denote by $F^{\mathcal{A}} \in \{0, 1\}$ its truth value in $\mathcal{A}$. A formula F is *satisfied* in $\mathcal{A}$, written $\mathcal{A} \models F$, if $F^{\mathcal{A}} = 1$. In this case, we also call $\mathcal{A}$ a *model* of F. We say that F is *satisfiable*, if F has a model. For two Σ-formulas F and G, we say F *entails* G, written $F \models G$, if $\mathcal{A} \models F$ implies $\mathcal{A} \models G$ for all Σ-interpretations $\mathcal{A}$.

Theories and Theory Combinations. A *Σ-theory* is a class of Σ-structures. Given a Σ-theory $\mathcal{T}$, a *$\mathcal{T}$-interpretation* is a Σ-interpretation $\mathcal{A}$ such that $\mathcal{A}|_{\Sigma} \in \mathcal{T}$. A Σ-formula is called *$\mathcal{T}$-satisfiable*, if it is satisfiable in some $\mathcal{T}$-interpretation. The *quantifier-free satisfiability problem of* $\mathcal{T}$ is to decide for every quantifier-free Σ-formula whether it is $\mathcal{T}$-satisfiable or not.

Let Σ be a signature with sorts S and $S' \subseteq S$. A Σ-theory is called *stably infinite* with respect to S', if each $\mathcal{T}$-satisfiable quantifier-free Σ-formula is satisfiable in a $\mathcal{T}$-interpretation $\mathcal{A}$ such that $s^{\mathcal{A}}$ has infinite cardinality, for all $s \in S'$.

Let $\mathcal{T}_i$ be a Σ_i-theory, for $i = 1, 2$, and let $\Sigma = \Sigma_1 \cup \Sigma_2$. The *combination* of $\mathcal{T}_1$ and $\mathcal{T}_2$ is the Σ-theory $\mathcal{T}_1 \oplus \mathcal{T}_2 = \{ \mathcal{A} \mid \mathcal{A}|_{\Sigma_1} \in \mathcal{T}_1 \text{ and } \mathcal{A}|_{\Sigma_2} \in \mathcal{A}_2 \}$. We call $\mathcal{T}_1 \oplus \mathcal{T}_2$ the *disjoint combination* of $\mathcal{T}_1$ and $\mathcal{T}_2$ if Σ_1 and Σ_2 are disjoint.

3 Logic of Graph Reachability and Stratified Sets

In this section we formally introduce the logic of graph reachability and stratified sets (GRASS), which is the target for our reduction of separation logic. Formulas in the logic are interpreted in (function) graphs and can express properties of the graph structure as well as properties of sets of nodes in the graph that are defined in terms of properties of the graph structure.

Syntax of GRASS. Throughout the rest of this paper we assume that $\mathcal{X}$ is a countably infinite set of variables of sorts node and set. We use the lower-case symbols $x, y \in \mathcal{X}$ for variables of sort node and upper-case symbols $X, Y \in \mathcal{X}$ for variables of sort set.

The syntax of GRASS is defined in Figure 1. A GRASS formula is a propositional combination of atoms. There are two types of atoms. Atoms of type A are either

$$T ::= x \mid h(T) \qquad A ::= T = T \mid T \xrightarrow{h\backslash T} T \quad R ::= A \mid \neg R \mid R \wedge R \mid R \vee R$$
$$S ::= X \mid \varnothing \mid S \backslash S \mid S \cap S \mid S \cup S \mid \{x.\, R\} \quad x \text{ does not occur below } h \text{ in } R$$
$$B ::= S = S \mid T \in S \qquad F ::= A \mid B \mid \neg F \mid F \wedge F \mid F \vee F$$

Fig. 1. Logic of graph reachability and stratified sets (GRASS)

equalities between terms of type T and *reachability predicates* of the form $t_1 \xrightarrow{h\backslash t_3} t_2$. The terms of type T represent nodes in the graph. They have associated sort node and are constructed from variables and application of the function symbol h, which represents the (functional) edge relation of the graph. Intuitively, a reachability predicate $t_1 \xrightarrow{h\backslash t_3} t_2$ is true if there exists a path in the graph that connects t_1 and t_2 without going through t_3. Atoms of type B are equalities between terms of type S, which have an associated sort set, and membership tests. Terms of type S represent *stratified sets* [36], i.e., their elements are interpreted – here, as nodes in the graph. S-terms include *set comprehensions* of the form $\{x.\, R\}$, where R is a Boolean combination of atoms of type A. We require that the term $h(x)$ does not occur in R. This side condition is important to ensure the decidability of the logic.

We use syntactic short-hands for implication, bi-implication, etc. We further write $t_1 \xrightarrow{h} t_2$ as an abbreviation for $t_1 \xrightarrow{h\backslash t_2} t_2$ and we use short-hands for the universal set $\mathcal{U}$, which stands for $\{x.\, x = x\}$, subset inclusion $S_1 \subseteq S_2$, which stands for $S_1 \cup S_2 = S_2$, and set enumerations $\{t_1, \ldots, t_n\}$, which stand for $\{x.\, x = t_1 \vee \cdots \vee x = t_n\}$ where x does not occur in $t_1, \ldots, t_n$. Finally, we write $X = Y \uplus Z$ for $X = Y \cup Z \wedge Y \cap Z = \varnothing$.

Example 1. Consider the formula $F \equiv Y = \{x.\, x \xrightarrow{h} y\} \wedge Z = \{x.\, x \xrightarrow{h} z\} \wedge \mathcal{U} = Y \uplus Z$. This formula describes all function graphs that consist of two disjoint connected components, one in which all nodes reach y, and one in which all nodes reach z.

Semantics of GRASS. We define the semantics of GRASS with respect to a specific theory $\mathcal{T}_{\mathsf{GS}}$. The theory $\mathcal{T}_{\mathsf{GS}}$ is the disjoint combination of a theory of reachability in function graphs $\mathcal{T}_{\mathsf{G}}$ and a theory of stratified sets $\mathcal{T}_{\mathsf{S}}$.

We first define the theory $\mathcal{T}_{\mathsf{G}}$. The structures in $\mathcal{T}_{\mathsf{G}}$ are over the signature $\Sigma_{\mathsf{G}} = (S_{\mathsf{G}}, \Omega_{\mathsf{G}}, \Pi_{\mathsf{G}})$ with sorts $S_{\mathsf{G}} = \{\mathsf{node}\}$, function symbols $\Omega_{\mathsf{G}} = \{h\}$, and predicate symbols $\Pi_H = \{\xrightarrow{h\backslash}\}$. The sort of h is $\mathsf{node} \to \mathsf{node}$ and the sort of $\xrightarrow{h\backslash}$ is $\mathsf{node} \times \mathsf{node} \times \mathsf{node}$. The structures in $\mathcal{T}_{\mathsf{G}}$ are defined as follows. For a binary relation r over a set S (respectively, a unary function from S to S), we denote by r^* the reflexive transitive closure of r. A structure $\mathcal{A}$ over signature Σ_{G} is in $\mathcal{T}_{\mathsf{G}}$ iff the following conditions are satisfied. First, the interpretation of the edge function h in $\mathcal{A}$ is constrained as follows: for all $u \in \mathsf{node}^{\mathcal{A}}$, the sets $\{\, v \in \mathsf{node}^{\mathcal{A}} \mid (u, v) \in (h^{\mathcal{A}})^* \,\}$ and $\{\, v \in \mathsf{node}^{\mathcal{A}} \mid (v, u) \in (h^{\mathcal{A}})^* \,\}$ are finite. Second, the interpretation of the reachability predicate is defined in terms of $h^{\mathcal{A}}$ as follows. For all $u, v, w \in \mathsf{node}^{\mathcal{A}}$

$$u \xrightarrow{h\backslash w}{}^{\mathcal{A}} v \iff (u, v) \in \{\, (u_1, h^{\mathcal{A}}(u_1)) \mid u_1 \in \mathsf{node}^{\mathcal{A}} \wedge u_1 \neq w \,\}^*$$

Next, we define the theory of stratified sets $\mathcal{T}_{\mathsf{S}}$ [36]. The structures in $\mathcal{T}_{\mathsf{S}}$ are over the signature $\Sigma_{\mathsf{S}} = (S_{\mathsf{S}}, \Omega_{\mathsf{S}}, \Pi_{\mathsf{S}})$ with sorts $S_{\mathsf{S}} = \{\mathsf{node}, \mathsf{set}\}$, function symbols

$$x, y \in \mathcal{X} \qquad \Sigma ::= x = y \mid x \neq y \mid x \mapsto y \mid \mathsf{ls}(x, y) \mid \Sigma * \Sigma \qquad H ::= \Sigma \mid \neg H \mid H \wedge H$$

Fig. 2. SLL$\mathbb{B}$: separation logic of linked lists

$\Omega_{\mathsf{S}} = \{\varnothing, \cap, \cup, \setminus\}$, and predicate symbols $\Pi_{\mathsf{S}} = \{\in\}$. The symbol $\varnothing$ is a constant of sort set and the function symbols $\cap$, $\cup$, and $\setminus$ all have sort $\mathsf{set} \times \mathsf{set} \to \mathsf{set}$. The predicate symbol $\in$ has sort $\mathsf{node} \times \mathsf{set}$. A structure $\mathcal{A}$ is in $\mathcal{T}_{\mathsf{S}}$ iff $\mathcal{A}$ interprets the sort set as the set of all subsets of $\mathsf{node}^{\mathcal{A}}$ and the symbols in Ω_{S} and Π_{S} are interpreted as expected.

Now define $\Sigma_{\mathsf{GS}} = \Sigma_{\mathsf{G}} \cup \Sigma_{\mathsf{S}}$ and $\mathcal{T}_{\mathsf{GS}} = \mathcal{T}_{\mathsf{G}} \oplus \mathcal{T}_{\mathsf{S}}$. We call the structures in $\mathcal{T}_{\mathsf{GS}}$ *heap structures*, referring to their use later on in the paper. Likewise, we call a Σ_{GS}-interpretation whose reduct is a heap structure a *heap interpretation*. We denote by $\mathcal{T}_{\mathsf{GS},\mathcal{X}}$ the set of all heap interpretations.

We next define the semantics of GRASS formulas. Let $\mathcal{A}$ be a heap interpretation. The evaluation of terms of type T and formulas of type R in $\mathcal{A}$ are defined as in first-order logic. Using these definitions we define the evaluation of a set comprehensions $\{x.\, R\}$ in $\mathcal{A}$ as follows: $\{x.\, R\}^{\mathcal{A}} = \{\, u \in \mathsf{node}^{\mathcal{A}} \mid R^{\mathcal{A}[x \mapsto u]} = 1 \,\}$.

The definition of the evaluation function is then extended by structural induction to terms of type S and formulas of type F, as expected. The notions of satisfiability and entailment are defined as for first-order logic, except that we restrict ourselves to heap interpretations, respectively, heap structures. We denote by $\mathcal{A} \models_{\mathsf{GS}} F$ that GRASS formula F is satisfied by heap interpretation $\mathcal{A}$.

The satisfiability problem of GRASS asks whether a given GRASS formula F is satisfiable. This problem is decidable. The decision procedure can be implemented within an SMT solver using a Nelson-Oppen combination of solvers for $\mathcal{T}_{\mathsf{G}}$ and $\mathcal{T}_{\mathsf{S}}$. We describe this procedure in the tech report. The following theorem only states its existence.

Theorem 2. *The satisfiability problem of GRASS is NP-complete.*

4 Separation Logic of Linked Lists

We consider separation logic formulas that are given by propositional combinations of formulas in separation logic of linked lists [5] (SLL). We refer to our fragment of separation logic simply as SLL$\mathbb{B}$. The syntax of the formulas in this fragment are given in Figure 2. That is, a formula is a propositional combination of *spatial formulas* Σ. A spatial formula is an equality or disequality of variables (of sort node), a *points-to predicate* $x \mapsto y$, a *list segment predicate* $\mathsf{ls}(x, y)$, or a *spatial conjunction* $\Sigma_1 * \Sigma_2$ of spatial formulas. We denote by $\mathcal{H}$ the set of all these formulas. We use syntactic sugar for disjunctions. We further write emp for $x = x$, false for $x \neq x$, and true for $\neg(x \neq x)$, where $x \in \mathcal{X}$ is some fixed variable.

The standard semantics of separation logic formulas is given with respect to a variable assignment (referred to as stack) and a partial function on memory addresses to values (referred to as the heap). In order to be able to easily relate formulas in SLL$\mathbb{B}$ and GRASS, we define the semantics of SLL$\mathbb{B}$ formulas in terms of heap interpretations

$$
\begin{array}{ll}
\mathcal{A}, X \models_{\mathsf{SL}} x = y & \text{iff } x^{\mathcal{A}} = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \varnothing \\
\mathcal{A}, X \models_{\mathsf{SL}} x \neq y & \text{iff } x^{\mathcal{A}} \neq y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \varnothing \\
\mathcal{A}, X \models_{\mathsf{SL}} x \mapsto y & \text{iff } h^{\mathcal{A}}(x^{\mathcal{A}}) = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \{x^{\mathcal{A}}\} \\
\mathcal{A}, X \models_{\mathsf{SL}} H_1 * H_2 & \text{iff } \exists U_1, U_2.\, U_1 \cup U_2 = X^{\mathcal{A}} \text{ and } U_1 \cap U_2 = \varnothing \text{ and} \\
 & \quad \mathcal{A}[X \mapsto U_1], X \models_{\mathsf{SL}} H_1 \text{ and } \mathcal{A}[X \mapsto U_2], X \models_{\mathsf{SL}} H_2 \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}(x, y) & \text{iff } \exists n \geqslant 0.\, \mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^n(x, y) \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^0(x, y) & \text{iff } x^{\mathcal{A}} = y^{\mathcal{A}} \text{ and } X^{\mathcal{A}} = \varnothing \\
\mathcal{A}, X \models_{\mathsf{SL}} \mathsf{ls}^{n+1}(x, y) & \text{iff } \exists u \in \mathsf{node}^{\mathcal{A}}.\, \mathcal{A}[z \mapsto u], X \models_{\mathsf{SL}} x \mapsto z * \mathsf{ls}^n(z, y) \\
 & \quad \text{and } x^{\mathcal{A}} \neq y^{\mathcal{A}} \text{ and } z \neq x \text{ and } z \neq y \\
\mathcal{A}, X \models_{\mathsf{SL}} H_1 \wedge H_2 & \text{iff } \mathcal{A}, X \models_{\mathsf{SL}} H_1 \text{ and } \mathcal{A}, X \models_{\mathsf{SL}} H_2 \\
\mathcal{A}, X \models_{\mathsf{SL}} \neg H & \text{iff not } \mathcal{A}, X \models_{\mathsf{SL}} H
\end{array}
$$

Fig. 3. Semantics of $\mathsf{SLL}\mathbb{B}$ in terms of heap interpretations

$\mathcal{A}$. Our semantics is consistent with the standard semantics (except for one minor deviation that we explain below). The interpretation of the edge function $h^{\mathcal{A}}$ plays the role of the heap in the standard semantics and the variable assignment in $\mathcal{A}$ plays the role of the stack. Since a heap structure $\mathcal{A}$ interprets h by a total function and the standard interpretation of separation logic is with respect to a heap that is a partial function, we must explicitly say which subset of $\mathsf{node}^{\mathcal{A}}$ we use to interpret a separation logic formula. For this purpose, we use a set variable $X \in \mathcal{X}$ whose interpretation in $\mathcal{A}$ determines this subset. We call $X^{\mathcal{A}}$ the *footprint* of the interpreted formula. The set variable X is a parameter of the semantics. The satisfaction relation is denoted by judgments of the form $\mathcal{A}, X \models_{\mathsf{SL}} H$, as defined in Figure 3.

If $\mathcal{A}, X \models_{\mathsf{SL}} H$ holds, we say that H is satisfied by $\mathcal{A}$ with respect to X, respectively, that $\mathcal{A}$ is a model of H with respect to X. Entailment between two $\mathsf{SLL}\mathbb{B}$ formulas H_1 and H_2 (written $H_1 \models_{\mathsf{SL}} H_2$) is then defined as expected.

The satisfiability problem for $\mathsf{SLL}\mathbb{B}$ asks whether a given $\mathsf{SLL}\mathbb{B}$ formula H is satisfiable in some heap interpretation $\mathcal{A}$ with respect to some set variable X. It follows from results in [15], that this problem is NP-complete.

Unlike the standard semantics of separation logic, our semantics is *precise* [13]. That is, the footprint of a spatial formula is uniquely defined in each model. In particular, (dis)equalities constrain the heap to be empty. For example, the formula $x \neq y \wedge \mathsf{ls}(x, y)$ is unsatisfiable because $x \neq y$ implies both that the heap is empty and that $\mathsf{ls}(x, y)$ is a non-empty list segment. On the other hand, the formula $x \neq y * \mathsf{ls}(x, y)$ is satisfiable and describes all heaps containing non-empty list segments from x to y, which is the meaning of $x \neq y \wedge \mathsf{ls}(x, y)$ in the standard semantics. The deviation from the standard semantics is therefore of little practical consequence and is in fact adopted by some separation logic tools. Our approach also works for the standard semantics of (dis)equalities, but the correctness proofs are more involved. We can further adapt our approach to handle other imprecise formulas such as formulas with disjunctions and

$$
\begin{aligned}
str_Y(x = y) &= (x = y,\ Y = \varnothing) \qquad str_Y(x \mapsto y) = (h(x) = y,\ Y = \{x\}) \\
str_Y(x \neq y) &= (x \neq y,\ Y = \varnothing) \qquad str_Y(\mathsf{ls}(x,y)) = (x \xrightarrow{h} y,\ Y = Btwn(x,y)) \\
str_Y(\Sigma_1 * \Sigma_2) &= \text{let } Y_1, Y_2 \in \mathcal{X} \text{ fresh and } (F_1, G_1) = tr_{Y_1}(\Sigma_1) \text{ and } (F_2, G_2) = tr_{Y_2}(\Sigma_2) \\
&\quad \text{in } (F_1 \wedge F_2 \wedge Y_1 \cap Y_2 = \varnothing,\ Y = Y_1 \cup Y_2 \wedge G_1 \wedge G_2) \\
tr_X(\Sigma) &= \text{let } Y \in \mathcal{X} \text{ fresh and } (F, G) = str_Y(\Sigma) \text{ in } (F \wedge X = Y,\ G) \\
tr_X(\neg H) &= \text{let } (F, G) = tr_X(H) \text{ in } (\neg F,\ G) \\
tr_X(H_1 \wedge H_2) &= \text{let } (F_1, G_1) = tr_X(H_1) \text{ and } (F_2, G_2) = tr_X(H_2) \\
&\quad \text{in } (F_1 \wedge F_2,\ G_1 \wedge G_2) \\
Tr_X(H) &= \text{let } (F, G) = tr_X(H) \text{ in } F \wedge G
\end{aligned}
$$

Fig. 4. Translation of SLL$\mathbb{B}$ to GRASS

conjunctions below spatial conjunction. The only problematic generalization that cannot be easily handled is to admit negation below spatial conjunction. To our knowledge, no (automated) SL tool supports such formulas because of the increased complexity.

5 Reduction of SLL$\mathbb{B}$ to GRASS

In the following, we present our reduction approach for automated reasoning about SLL$\mathbb{B}$ formulas. We show that every SLL$\mathbb{B}$ formula can be reduced in linear time to an equisatisfiable GRASS formula. By using our decision procedure for GRASS, this reduction yields an SMT-based decision procedure for the satisfiability and entailment problem of SLL$\mathbb{B}$. Furthermore, it enables theory combination of SLL$\mathbb{B}$ with signature disjoint theories within the Nelson-Oppen combination framework.

Translating SLL$\mathbb{B}$ to GRASS. We start with the translation function Tr that maps SLL$\mathbb{B}$ formulas to GRASS formulas. It is shown in Figure 4. The function is parameterized by a set variable X, which holds the footprint of the translated formula. The translation is defined using two auxiliary functions str and tr.

The function str maps a set variable Y and a spatial formula Σ to a pair of GRASS formulas (F, G). The formula F captures the structure of Σ, while the formula G defines auxiliary set variables that are used to link Y to the footprint of Σ. The function str is defined recursively on the structure of Σ. Note that it closely follows the semantics of spatial formulas. In particular, to define the footprint Y of a spatial conjunction $\Sigma_1 * \Sigma_2$, the function str introduces two fresh set variables to capture the footprints of Σ_1 and Σ_2, respectively, and then defines Y as the disjoint union of these two sets. Also, note that we do not need induction to translate list segments. Instead, the structure and footprint of a list segment are translated directly using reachability predicates. Here, we write $Btwn(x, y)$ as a short-hand for the set comprehension $\{z.x \xrightarrow{h \setminus y} z \wedge z \neq y\}$.

The function tr translates Boolean combinations of spatial conjunctions. At the leaf level, tr introduces fresh set variables Y to translate the meaning of spatial formulas Σ and asserts $X = Y$ in the structural constraint. The constraints G defining the auxiliary

set variables are propagated to the top level where the function Tr conjoins them with the structural constraint F of the entire formula. The following lemma implies that the translation yields an equisatisfiable formula.

Lemma 3. *Let H be an SLL$\mathbb{B}$ formula, $X \in \mathcal{X}$, and $\mathcal{A}$ a heap interpretation. Then $\mathcal{A}$ satisfies H with respect to X iff there exist subsets $U_1, \ldots, U_n$ of $\mathsf{node}^{\mathcal{A}}$ such that $\mathcal{A}[Y_1 \mapsto U_1, \ldots, Y_n \mapsto U_n] \models_{\mathsf{GS}} Tr_X(H)$, where $Y_1, \ldots, Y_n$ are the fresh set variables introduced in the translation $Tr_X(H)$.*

Note that the auxiliary set variables Y_i that are introduced for the translation of spatial conjunctions are implicitly existentially quantified. Hence, when a spatial conjunction appears below an odd number of negations, these existential quantifiers should become universal quantifiers. One might therefore wonder why the propagation of constraints G to the top level of the formula is still correct, since all set variables remain existentially quantified. It is here where the precise semantics of spatial formulas helps. Each constraint G is a conjunction of equalities defining the sets Y_i as finite unions of set comprehensions. Therefore, these constraints are satisfiable in any given heap interpretation. In fact, for each constraint G and heap interpretation $\mathcal{A}$, there exists exactly one assignment of the Y_i to $U_i \subseteq \mathsf{node}^{\mathcal{A}}$ that makes G true in $\mathcal{A}$. Hence, the formulas $\exists Y_1, \ldots, Y_n. F \wedge G$ and $\forall Y_1, \ldots, Y_n. G \Rightarrow F$ are equivalent.

For two SLL$\mathbb{B}$ formulas H_1 and H_2, we have that H_1 entails H_2 iff $H_1 \wedge \neg H_2$ is unsatisfiable. It follows from Lemma 3 that our translation yields a decision procedure for satisfiability and entailment of SLL$\mathbb{B}$ formulas.

Theorem 4. *The satisfiability and entailment problems of SLL$\mathbb{B}$ are reducible in linear time to the satisfiability problem of GRASS.*

Example 5. Consider the two separation logic formulas $H_1 \equiv x \neq z * x \mapsto y * \mathsf{ls}(y, z)$, and $H_2 \equiv \mathsf{ls}(x, z)$. Both formulas describe heaps consisting of an acyclic list segment from x to z. In the case of H_1, the segment is non-empty, while H_2 also allows the empty segment, i.e., $H_1 \models_{\mathsf{SL}} H_2$. Let $X \in \mathcal{X}$ be a set variable. Then $Tr_X(H_1)$ is

$$x \neq z \wedge h(x) = y \wedge y \xrightarrow{h} z \wedge Y_2 \cap Y_3 = \varnothing \wedge Y_4 \cap Y_5 = \varnothing \wedge X = Y_1 \wedge Y_1 = Y_2 \cup Y_3 \wedge Y_2 = \varnothing \wedge Y_3 = Y_4 \cup Y_5 \wedge Y_4 = \{x\} \wedge Y_5 = Btwn(y, z)$$

which can be simplified to $x \neq z \wedge h(x) \xrightarrow{h} z \wedge X = \{x\} \uplus Btwn(h(x), z)$. We further have $Tr_X(\neg H_2) \equiv \neg(x \xrightarrow{h} z \wedge X = Y_6) \wedge Y_6 = Btwn(x, z)$. To see why $Tr_X(H_1) \wedge Tr_X(\neg H_2)$ is unsatisfiable, note that $h(x) \xrightarrow{h} z$ implies $x \xrightarrow{h} z$ and $Btwn(x, z) = \{x\} \cup Btwn(h(x), z)$.

Combining SLL$\mathbb{B}$ with Other Theories. We next show that the theory $\mathcal{T}_{\mathsf{GS}}$ behaves well with respect to theory combination. For instance, we can combine it with a theory of integer arithmetic for interpreting memory addresses. We can then use this theory to reason about SL fragments in which we allow address arithmetic. Similar combinations enable reasoning about fragments that can express properties about data. To implement these theory combinations, we can leverage the Nelson-Oppen combination framework [23] provided in SMT solvers.

Formally, let Σ_{node} be a signature that is disjoint from Σ_{GS} and that contains at least the sort node. Let further $\mathcal{T}_{\mathsf{node}}$ be a decidable Σ_{node}-theory that is stably infinite with respect to sort node. For example, $\mathcal{T}_{\mathsf{node}}$ may be the theory of linear arithmetic, interpreting the sort node as integers. Define $\Sigma = \Sigma_{\mathsf{node}} \cup \Sigma_{\mathsf{GS}}$ and $\mathcal{T} = \mathcal{T}_{\mathsf{node}} \oplus \mathcal{T}_{\mathsf{GS}}$.

We show that our reduction of SLL$\mathbb{B}$ to GRASS allows us to decide satisfiability of conjunctions $H \wedge G$ of SLL$\mathbb{B}$ formulas H with quantifier-free Σ_{node}-formulas G. Such conjunctions are interpreted in Σ-interpretations, as expected. Given such a conjunction $H \wedge G$, the decision procedure checks $\mathcal{T}$-satisfiability of the formula $\mathit{reduce}(\mathit{Tr}_X(H)) \wedge G$, where $X \in \mathcal{X}$ does not appear in G. This check is implemented using a Nelson-Oppen combination of the decision procedure for $\mathcal{T}_{\mathsf{GS}}$ and the decision procedure for $\mathcal{T}_{\mathsf{node}}$. To show the completeness of this combination procedure, let $\mathcal{T}_{\mathsf{SL}}$ be the Σ_{GS}-theory defined as follows:

$$\mathcal{T}_{\mathsf{SL}} = \{\, \mathcal{A}|_{\Sigma_{\mathsf{GS}}} \mid \exists H \in \mathcal{H}, \mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}, X \in \mathcal{X}.\ \mathcal{A}, X \models_{\mathsf{SL}} H \,\}$$

We call $\mathcal{T}_{\mathsf{SL}}$ the theory of SLL$\mathbb{B}$. Completeness then follows from the following theorem.

Theorem 6. *The theory $\mathcal{T}_{\mathsf{SL}}$ is stably infinite with respect to sort* node.

Theorem 6 follows from the fact that the theory of the fragment of GRASS that is defined by the translation function Tr is stably infinite with respect to sort node. Incidentally, this is not true for the full theory of GRASS. For example, the GRASS formula $\{x.\, x = y\} = \mathcal{U}$ has only models where the interpretation of sort node has cardinality 1. Theory combination for the full theory of GRASS is still possible using a more complex combination procedure that requires GRASS to be extended with linear cardinality constraints [34].

6 Extensions

In this section, we describe several extensions of GRASS to support symbolic execution of programs on GRASS formulas and more expressive separation logic fragments.

Arrays. One advantage of our approach is that it enables the use of separation logic in existing verification tools that already provide backends to SMT solvers, without requiring specialized symbolic execution engines for separation logic. However, we then need a form of symbolic execution for GRASS formulas that is supported by existing tools. In particular, the logic must be able to express the effect of heap updates concisely. We can do this by extending GRASS with a theory of arrays to represent mutable data structure fields. That is, we model fields as arrays whose indices and elements are of sort node. For this purpose, we extend the signature Σ_{GS} with an additional sort field, and additional function symbols $\mathsf{sel} : \mathsf{field} \times \mathsf{node} \to \mathsf{node}$ and $\mathsf{upd} : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \to \mathsf{node}$ to model field reads and writes. Also, the reachability predicate will now be of the form $\bullet \xrightarrow{\bullet\backslash\bullet} \bullet : \mathsf{field} \times \mathsf{node} \times \mathsf{node} \times \mathsf{node}$, taking a field as additional parameter. It follows from results in [32] that the quantifier-free satisfiability problem for this extension remains decidable in NP. In the tech report [28], we show how to use this extension to decide validity of verification conditions with SL assertions.

Beyond Singly-Linked Lists. To support SL fragments with inductive predicates for more diverse data structures, we consider several non-disjoint extensions of GRASS and then extend the translation function for the additional inductive predicates appropriately. For example, suppose we consider heap structures for list nodes consisting of two fields: a pointer n to the next node in the list and a data field d storing an integer value. Now suppose we want to extend the SLL$\mathbb{B}$ with an inductive predicate $\mathsf{sls}(x, y)$ representing a heap region consisting of a list segment from x to y, whose data values are sorted. Automated reasoning about formulas with such a predicate is more difficult to achieve using conventional SL provers because the predicate relates the memory layout with constraints on the stored data. We can easily support such a predicate in our approach by relying on the capabilities of the underlying SMT solver. To extend our translation from Section 5 to the sorted list predicate it suffices to define:

$$str_Y(\mathsf{sls}(x, y)) = (x \xrightarrow{n} y \wedge \forall z, w \in Y.\, z \xrightarrow{n} w \Rightarrow d(z) \leqslant d(w),\ Y = Btwn(x, y))$$

Under mild assumptions, it follows from results in [22] that the quantified constraint expressing the sortedness property is a local theory extension [31] and remains in a decidable fragment. This allows us to reduce reasoning about sorted lists to reasoning in a disjoint combination of $\mathcal{T}_{\mathsf{GS}}$ with the theory of free function symbols (for the data field) and the theory of linear arithmetic. Similar reductions can be given for predicates encoding data structures with more complex linking patterns, such as doubly-linked lists, lists with head pointers, nested lists, etc. For example, the translation for the usual doubly-linked list predicate $\mathsf{dlls}(x, a, y, b)$ over forward pointer field n and backward pointer field p (see, e.g. [4]) is as follows:

$$\begin{aligned} str_Y(\mathsf{dlls}(x, a, y, b)) = (\, & x \xrightarrow{n} y \wedge (x = y \wedge a = b \vee p(x) = a \wedge n(b) = y \wedge b \in Y) \wedge \\ & \forall z \in Y.\, n(z) \in Y \Rightarrow p(n(z)) = z,\ Y = Btwn(x, y)\ ) \end{aligned}$$

The quantified constraint in the translation again constitutes a local theory extension that remains decidable and can be handled efficiently. One can also provide translations for inductive predicates describing tree data structures by using an appropriate first-order theory for reachability in trees, such as the one presented in [33].

7 Frame Inference and Abduction

Many operations in SL-based program analyses, including the application of the frame rule, involve more general forms of entailment tests referred to as frame inference [7,18] and abduction [11]. The *frame inference problem* is to compute for a pair of SLL$\mathbb{B}$ formulas (H, G), a formula F such that $H \models_{\mathsf{SL}} G * F$ holds, if such F exists. We call F the *frame* and we denote such frame inference problems by $H \models_{\mathsf{SL}} G * F?$. Likewise, the *abduction problem* is to find an *anti-frame* F for (H, G) such that $H * F \models_{\mathsf{SL}} G$. We denote abduction problems by $H * F? \models_{\mathsf{SL}} G$. In the following, we explain how to solve frame inference and abduction problems using our decision procedure for GRASS in combination with a model-generating SMT solver.

Inverse Translation. Our technique for frame inference and abduction uses a characterization of a GRASS formula F in terms of a finite set of partial interpretations

$\mathsf{PMod}_X(F)$ that we obtain from the models of F, where X is some set variable occurring in F. We use this set of partial models to define an *inverse translation function* that maps F to an $\mathsf{SLL}\mathbb{B}$ formula $Tr_X^{-1}(F)$. The purpose of the set variable X is to carve out a specific partial substructures of each model of F that is then captured by $Tr_X^{-1}(F)$.

We start by defining $\mathsf{PMod}_X(F)$. Let F be a GRASS formula and let $X \in \mathcal{X}$ be a set variable. Let further $N \subseteq \mathcal{X}$ be the set of all free variables of sort node in F and define $V = N \cup \{X\}$. For $\mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}$, define $N^{\mathcal{A}} = \{ x^{\mathcal{A}} \mid x \in N \}$. Further, define $\mathcal{A}_{F,X}$ as the partial Σ_{GS}-interpretation for variables V that is obtained from $\mathcal{A}$ by defining $\mathsf{node}^{\mathcal{A}_{F,X}} = N^{\mathcal{A}}$ and restricting the interpretation of all symbols $\Omega_{\mathsf{GS}} \cup \Pi_{\mathsf{GS}} \cup V$ in $\mathcal{A}$ to $N^{\mathcal{A}}$. We then define $\mathsf{PMod}_X(F) = \{ \mathcal{A}_{F,X} \mid \mathcal{A} \models_{\mathsf{GS}} F \}$.

To simplify the presentation, we restrict ourselves to a specific class of GRASS formulas: we say that F is *X-closed*, if for all $\mathcal{B} \in \mathsf{PMod}_X(F)$ and $u \in X^{\mathcal{B}}$, $h^{\mathcal{B}}(u)$ is defined or there exists some $v \in \mathsf{node}^{\mathcal{B}}$ such that $v \neq u$ and $u \rightarrow^{\mathcal{B}} v$. In the following, we assume that F is X-closed. The inverse translation can be generalized to arbitrary GRASS formulas. However, this requires the introduction of additional Skolem constants (respectively, explicit existential quantifiers in $\mathsf{SLL}\mathbb{B}$).

Let $\mathcal{B} \in \mathsf{PMod}(F)$. Define a partial function $succ^{\mathcal{B}} : \mathsf{node}^{\mathcal{B}} \rightharpoonup \mathsf{node}^{\mathcal{B}}$ as follows. For all $u \in \mathsf{node}^{\mathcal{B}}$, let $succ^{\mathcal{B}}(u) = v$, where $v \in \mathsf{node}^{\mathcal{B}}$ is the unique node such that $v \neq u$ and for all $w \in \mathsf{node}^{\mathcal{B}}$, if $w \neq u$ and $u \rightarrow^{\mathcal{B}} w$, then $u \xrightarrow{h \setminus w}{}^{\mathcal{B}} v$, if such a node v exists. Otherwise, $succ^{\mathcal{B}}(u)$ is undefined. For every $u \in \mathsf{node}^{\mathcal{B}}$, let $x_u \in N$ such that $x_u^{\mathcal{B}} = u$. Now define a spatial conjunction $tr_X^{-1}(\mathcal{B})$ of $\mathsf{SLL}\mathbb{B}$ atoms as follows. First, for all distinct $x, y \in N$, if $x^{\mathcal{B}} = y^{\mathcal{B}}$, then $tr_X^{-1}(\mathcal{B})$ contains the spatial conjunct $x = y$, otherwise it contains $x \neq y$. Second, for every $u \in X^{\mathcal{B}}$, $tr_X^{-1}(\mathcal{B})$ contains a spatial conjunct Σ_u defined as follows: if $h^{\mathcal{B}}(u) = v$ for some $v \in N^{\mathcal{A}}$, then $\Sigma_u = (x_u \mapsto x_v)$; otherwise, $\Sigma_u = \mathsf{ls}(x_u, x_v)$ where v is such that $succ^{\mathcal{B}}(u) = v$.

Note that the set $\mathsf{PMod}(F)$ is finite up to isomorphism. If it is empty, we define $Tr_X^{-1}(F) = \mathsf{false}$. Otherwise, let $\mathcal{B}_1, \ldots, \mathcal{B}_n$ be representatives of all isomorphism classes of $\mathsf{PMod}(F)$. Then define $Tr_X^{-1}(F) = tr_X^{-1}(\mathcal{B}_1) \vee \ldots \vee tr_X^{-1}(\mathcal{B}_n)$.

The following lemma states the correctness of this inverse translation function.

Lemma 7. *Let $X \in \mathcal{X}$ and F be an X-closed GRASS formula. Then for all $\mathcal{A} \in \mathcal{T}_{\mathsf{GS},\mathcal{X}}$:*

1. *if $\mathcal{A} \models_{\mathsf{GS}} F$, then $\mathcal{A}, X \models_{\mathsf{SL}} Tr_X^{-1}(F)$, and*
2. *if $\mathcal{A}, X \models_{\mathsf{SL}} Tr_X^{-1}(F)$, then $\mathcal{A}_{F,X} \in \mathsf{PMod}_X(F)$.*

Note that we can compute $\mathsf{PMod}_X(F)$ by solving the All-SAT problem for F using a model-generating SMT solver that implements the decision procedure for $\mathcal{T}_{\mathsf{GS}}$. From each model $\mathcal{A}$ of F that is generated by the solver, we compute the partial model $\mathcal{A}_{F,X}$. This partial model then serves as a blocking clause for the solver to eliminate all models of F from the search space that are mapped to the isomorphism class of $\mathcal{A}_{F,X}$. If we apply this technique without further optimizations, then the computed set $\mathsf{PMod}_X(F)$ (and hence the formula $Tr_X^{-1}(F)$) will be (worst-case) exponential in the size of F. This is because each partial model fixes an arrangement of equalities between the variables in N. The enumeration process can be improved by generalizing each computed partial model before it is further processed, e.g., by dropping inequalities that are not implied by F. Only the generalized partial models are then used as blocking clauses, respectively, in the inverse translation function.

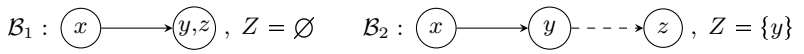


Fig. 5. The set $\mathsf{PMod}_Z(F)$ for formula F in Example 8

Example 8. Consider the GRASS formula

$$F \equiv x \neq z \wedge x \xrightarrow{h} z \wedge h(x) = y \wedge X = Btwn(x, z) \wedge Y = \{x\} \wedge Z = X \backslash Y$$

The set X contains all nodes on the path from x to z, excluding z. The path exists because of $x \xrightarrow{h} z$. Hence Z contains all nodes in X except for x itself. The set $\mathsf{PMod}_Z(F)$ consists of two isomorphism classes of partial models represented by $\mathcal{B}_1$ and $\mathcal{B}_2$ depicted in Figure 5. The solid edges denote the interpretation of h, the dashed edges denote the partial function $succ^{\mathcal{B}_i}$ for the nodes on which h is undefined. The partial models show that F is Z-closed. We then have $tr_Z^{-1}(\mathcal{B}_1) = x \neq z * x \neq y * y = z$ and $tr_Z^{-1}(\mathcal{B}_2) = x \neq z * x \neq y * y \neq z * \mathsf{ls}(y, z)$. From this we obtain $Tr_Z^{-1}(F) = tr_Z^{-1}(\mathcal{B}_1) \vee tr_Z^{-1}(\mathcal{B}_2) \equiv x \neq z * x \neq y * \mathsf{ls}(y, z)$.

Solving Frame Inference and Abduction Problems. We now show how we use the inverse translation function to solve frame inference problems. This technique can then be easily adapted for abduction.

A formula $H \in \mathcal{H}$ is called *positive* if it does not contain negations. For a positive formula H, we always have that $Tr_X(H)$ is X-closed. To ensure that we can use the inverse translation function from the previous section, we therefore restrict ourselves to frame inference problems in the positive fragment of SLL$\mathbb{B}$.

Let H and G be two positive SLL$\mathbb{B}$ formulas and suppose that $H \models_{\mathsf{SL}} G * F?$ has a solution. To compute a solution, define the GRASS formula

$$Frame_Z(H, G) = Tr_X(H) \wedge Tr_Y(G) \wedge Z = X \backslash Y$$

where $X, Y, Z \in \mathcal{X}$ are distinct set variables. Note that, the set variable Z describes the footprint of the frame. Moreover, the formula $Frame_Z(H, G)$ is Z-closed. Hence, the SLL$\mathbb{B}$ formula $Tr_Z^{-1}(Frame_Z(H, G))$ is a valid frame for (H, G).

It remains to check whether $H \models_{\mathsf{SL}} G * F?$ has a solution. For this purpose, define the GRASS formula

$$NoFrame(H, G) = Tr_X(H) \wedge Trf_X(\neg G)$$

where $X \in \mathcal{X}$ and Trf is like Tr, except that the constraints $X = Y$ in the case for $tr_X(\Sigma)$ are replaced by $Y \subseteq X$. Then $H \models_{\mathsf{SL}} G * F?$ has a solution iff $NoFrame(H, G)$ is unsatisfiable.

In order to adapt this technique for solving abduction problems $H * F? \models_{\mathsf{SL}} G$, it suffices to replace $Z = X \backslash Y$ in $Frame_Z(H, G)$ by $Z = Y \backslash X$, and the constraints $Y \subseteq X$ in $NoFrame(H, G)$ by $X \subseteq Y$.

8 Implementation and Experiments

We have implemented our decision procedure for GRASS together with the translation of SLL$\mathbb{B}$ in a prototype prover. We have further developed a verification tool called GRASShopper that builds on top of this prover[2]. Currently we use Z3 [16] as the underlying SMT solver because our implementation relies on Z3's model-based quantifier instantiation mechanism (MBQI).

To decide satisfiability of a GRASS formula F, we generate an equisatisfiable Σ_{GS}-formula G, which we then check for $\mathcal{T}_{\mathsf{GS}}$-satisfiability. We have not yet implemented a dedicated solver for the theory of graph reachability $\mathcal{T}_{\mathsf{G}}$. Instead, we use the finite first-order axiomatizations of this theory that are described in [22, 32]. To decide satisfiability of G, we conjoin the theory axioms with G and then partially instantiate quantified variables in the resulting formula with ground terms occurring in G. We only instantiate variables that occur below function symbols in the axioms of $\mathcal{T}_{\mathsf{G}}$. This keeps the size of the formulas that are given to the SMT solver reasonably small. The partial instantiation is guaranteed to be complete because $\mathcal{T}_{\mathsf{G}}$ is a local theory [31]. Details about this result can also be found in [32]. The partially instantiated axioms are in the EPR fragment of first-order logic (aka the Bernays-Schönfinkel-Ramsey class). The EPR fragment can be decided quite efficiently using Z3's MBQI mechanism. Stratified sets can be encoded directly in Z3 using combinatory array logic [17]. However, according to the Z3 developers, the array theory does currently not behave well with MBQI. We therefore also partially instantiate the axioms of stratified sets to remain in the EPR fragment.

Our tool GRASShopper uses the prover to verify list-manipulating programs written in a simple imperative language. The programs are expected to be annotated with procedure contracts and loop invariants expressed in separation logic. Each procedure is verified in isolation. To handle loops and procedure calls efficiently, the tool implements a frame rule that avoids explicit inference of frames. Instead, we encode frames implicitly in the formula that is given to the SMT solver. More details about this implementation can be found in the technical report. Currently, GRASShopper supports singly, doubly-linked, and sorted list predicates. We are planing to add support for user-defined predicates in the future. Since our prover yields a decision procedure for checking the generated verification conditions, we use the SMT solver to produce counterexamples for faulty programs, which our tool can then visualize.

We have applied our prototype to verify partial correctness specifications (including absence of run-time errors) of typical list-manipulating programs, including sorting algorithms. The considered programs contain loops and (recursive) procedure calls. Some of the programs consist of multiple procedures. Table 1 shows the results of the experiments. For example, the program "pairwise sum" takes two sorted lists as input and creates a new list whose entries are the pairwise sums of the entries in the input lists. We then show that the resulting list is again sorted. For the sorting algorithms, we proved that the output list is sorted but we did not check that it is a permutation of the input list. To verify the programs manipulating doubly-linked and sorted lists we used a Nelson-Oppen combination of $\mathcal{T}_{\mathsf{GS}}$ with the theory of equality and uninterpreted function symbols, and the theory of linear arithmetic.

[2] The tool is available at `http://cs.nyu.edu/wies/software/grasshopper`.

Table 1. Experimental results for the verification of list-manipulating programs. The columns marked "sl" refer to singly-linked list versions of the benchmarks, the "dl" columns to doubly-linked lists, and the "rec sl" columns to recursive implementations with singly-linked lists. Finally, the columns "sls" refer to sorted singly-linked lists. The columns "#" give the number of queries to the SMT solver and the "t" columns refer to the total running time in seconds.

program	sl		dl		rec sl		sls		program	sl		dl		rec sl		sls	
	#	t	#	t	#	t	#	t		#	t	#	t	#	t	#	t
concat	4	0.1	5	1.3	6	0.6	5	0.2	insert	6	0.2	5	1.5	5	0.2	6	0.4
copy	4	0.2	4	3.9	6	0.8	7	3.5	reverse	4	0.1	4	0.5	6	0.2	4	0.2
filter	7	0.6	5	1.1	8	0.4	5	1.1	remove	8	0.2	8	0.8	7	0.2	7	0.5
free	5	0.1	5	0.3	4	0.1	5	0.1	traverse	4	0.1	5	0.3	3	0.1	4	0.2
insertion sort							10	0.7	double all							7	2.2
merge sort							25	24	pairwise sum							10	20

9 Conclusions

We presented a reduction of decidable separation logic fragments to a decidable first-order logic fragment called GRASS. Our reduction enables the seamless integration of an SL prover into an SMT solver, which has promising applications in program verification. We demonstrated the feasibility of our approach using a prototype implementation. Future directions include the development of dedicated theory solvers for graph reachability and stratified sets, which underlie the decision procedure for GRASS.

References

1. Banerjee, A., Naumann, D.A., Rosenberg, S.: Regional logic for local reasoning about global invariants. In: Vitek, J. (ed.) ECOOP 2008. LNCS, vol. 5142, pp. 387–411. Springer, Heidelberg (2008)
2. Barnett, M., Chang, B.-Y.E., DeLine, R., Jacobs, B., Leino, K.R.M.: Boogie: A modular reusable verifier for object-oriented programs. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 364–387. Springer, Heidelberg (2006)
3. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)
4. Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P., Wies, T., Yang, H.: Shape analysis for composite data structures. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 178–192. Springer, Heidelberg (2007)
5. Berdine, J., Calcagno, C., O'Hearn, P. W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
6. Berdine, J., Calcagno, C., O'Hearn, P.W.: Smallfoot: Modular automatic assertion checking with separation logic. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 115–137. Springer, Heidelberg (2006)
7. Berdine, J., Calcagno, C., O'Hearn, P.W.: Symbolic execution with separation logic. In: Yi, K. (ed.) APLAS 2005. LNCS, vol. 3780, pp. 52–68. Springer, Heidelberg (2005)

8. Berdine, J., Cook, B., Ishtiaq, S.: SLAyer: Memory Safety for Systems-Level Code. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 178–183. Springer, Heidelberg (2011)
9. Bobot, F., Filliâtre, J.-C.: Separation predicates: a taste of separation logic in first-order logic. In: Aoki, T., Taguchi, K. (eds.) ICFEM 2012. LNCS, vol. 7635, pp. 167–181. Springer, Heidelberg (2012)
10. Bouajjani, A., Dragoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012)
11. Calcagno, C., Distefano, D., O'Hearn, P.W., Yang, H.: Compositional shape analysis by means of bi-abduction. In: POPL (2009)
12. Calcagno, C., Gardner, P., Hague, M.: From separation logic to first-order logic. In: Sassone, V. (ed.) FOSSACS 2005. LNCS, vol. 3441, pp. 395–409. Springer, Heidelberg (2005)
13. Calcagno, C., O'Hearn, P.W., Yang, H.: Local action and abstract separation logic. In: LICS, pp. 366–378. IEEE Computer Society (2007)
14. Chatterjee, S., Lahiri, S.K., Qadeer, S., Rakamarić, Z.: A reachability predicate for analyzing low-level software. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 19–33. Springer, Heidelberg (2007)
15. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
16. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
17. de Moura, L., Bjørner, N.: Generalized, efficient array decision procedures. In: FMCAD, pp. 45–52. IEEE (2009)
18. Distefano, D., Parkinson, M.J.: jStar: towards practical verification for Java. In: OOPSLA, pp. 213–226. ACM (2008)
19. Dudka, K., Peringer, P., Vojnar, T.: Predator: A practical tool for checking manipulation of dynamic data structures using separation logic. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 372–378. Springer, Heidelberg (2011)
20. Jacobs, B., Smans, J., Philippaerts, P., Vogels, F., Penninckx, W., Piessens, F.: VeriFast: A Powerful, Sound, Predictable, Fast Verifier for C and Java. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 41–55. Springer, Heidelberg (2011)
21. Kassios, I.T.: The dynamic frames theory. Formal Asp. Comput. 23(3), 267–288 (2011)
22. Lahiri, S.K., Qadeer, S.: Back to the future: revisiting precise program verification using SMT solvers. In: POPL, pp. 171–182 (2008)
23. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM TOPLAS 1(2), 245–257 (1979)
24. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001 and EACSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
25. Parkinson, M.J., Summers, A.J.: The relationship between separation logic and implicit dynamic frames. Logical Methods in Computer Science 8(3) (2012)
26. Pérez, J.A.N., Rybalchenko, A.: Separation logic + superposition calculus = heap theorem prover. In: PLDI, pp. 556–566. ACM (2011)
27. Pérez, J.A.N., Rybalchenko, A.: Separation Logic Modulo Theories. Technical Report arXiv:1303.2489, arXiv.org (2013)
28. Piskac, R., Wies, T., Zufferey, D.: Automating Separation Logic using SMT. Technical Report TR2013-954, New York University (2013)

29. Ranise, S., Ringeissen, C., Zarba, C.G.: Combining data structures with nonstably infinite theories using many-sorted logic. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 48–64. Springer, Heidelberg (2005)
30. Rosenberg, S., Banerjee, A., Naumann, D.A.: Decision procedures for region logic. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 379–395. Springer, Heidelberg (2012)
31. Sofronie-Stokkermans, V.: Hierarchic reasoning in local theory extensions. In: Nieuwenhuis, R. (ed.) CADE 2005. LNCS (LNAI), vol. 3632, pp. 219–234. Springer, Heidelberg (2005)
32. Totla, N., Wies, T.: Complete instantiation-based interpolation. In: POPL. ACM (2013)
33. Wies, T., Muñiz, M., Kuncak, V.: An efficient decision procedure for imperative tree data structures. In: Bjørner, N., Sofronie-Stokkermans, V. (eds.) CADE 2011. LNCS, vol. 6803, pp. 476–491. Springer, Heidelberg (2011)
34. Wies, T., Piskac, R., Kuncak, V.: Combining theories with shared set operations. In: Ghilardi, S., Sebastiani, R. (eds.) FroCoS 2009. LNCS, vol. 5749, pp. 366–382. Springer, Heidelberg (2009)
35. Yang, H., Lee, O., Berdine, J., Calcagno, C., Cook, B., Distefano, D., O'Hearn, P.W.: Scalable shape analysis for systems code. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 385–398. Springer, Heidelberg (2008)
36. Zarba, C.G.: Combining sets with elements. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 762–782. Springer, Heidelberg (2004)

SeLoger: A Tool for Graph-Based Reasoning in Separation Logic

Christoph Haase[1,⋆], Samin Ishtiaq[2],
Joël Ouaknine[3], and Matthew J. Parkinson[2]

[1] LSV – CNRS & ENS Cachan, France
[2] Microsoft Research Cambridge, UK
[3] Department of Computer Science, University of Oxford, UK

Abstract. This paper introduces the tool SeLoger, which is a reasoner for satisfiability and entailment in a fragment of separation logic with pointers and linked lists. SeLoger builds upon and extends graph-based algorithms that have recently been introduced in order to settle both decision problems in polynomial time. Running SeLoger on standard benchmarks shows that the tool outperforms current state-of-the-art tools by orders of magnitude.

1 Introduction

Tools based on separation logic [4,6] have shown tremendous promise when applied to the problem of formal verification in the presence of mutable data-structures. For example, shape analysis tools such as SpaceInvader, Thor, SLAyer or Infer are nowadays being applied to a range of low-level industrial systems code. Inside, these shape analysis tools mix traditional abstract interpretation techniques (*e.g.* custom abstract joins) combined with entailment procedures for restricted subsets of separation logic. Thus, one method for improving the scalability and applicability of these tools is to improve the underlying entailment or other decision procedures. This has been an active area of recent research, see *e.g.* [2,3,5].

Recently, we have shown in [3] that entailment in the fragment of separation logic with pointers and linked lists can be decided in polynomial time. This fragment was introduced in [1], and it forms the basis of a number of tools such as Smallfoot, SpaceInvader, and SLAyer. Traditionally, the separation logic reasoners integrated in those tools decide entailment via a syntactic proof search. In contrast, the decision procedure presented in [3] takes a different approach which is based on graph-theoretical methods.

In this paper, we introduce the tool SeLoger (SEparation LOgic GraphbasEd Reasoner) which implements an extension of the decision procedures presented in [3]. In Section 4, we compare SeLoger to the tool SLP by

⋆ Parts of the research were carried out while the author was an intern at Microsoft Research Cambridge, UK, and while the author held an EPSRC PhD+ fellowship at the Department of Computer Science, University of Oxford, UK. The author is supported by the French Agence Nationale de la Recherche, ReacHard (grant ANR-11-BS02-001).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 790–795, 2013.

Navarro Peréz and Rybalchenko [5]. They show that SLP outperforms the reasoners in Smallfoot and jStar by several orders of magnitude, and we can show that SeLoger outperforms SLP by orders of magnitude.

Recently, in [2] Bouajjani *et al.* have introduced the tool SLAD which also builds upon some of the ideas presented in [3]. One difference to our tool is that it decides entailment under *intuitionistic* semantics, which is also the semantic model considered in [3]. In contrast, the semantic model dealt with in [1] is *non-intuitionistic*, and the decision procedure implemented in SeLoger extends the one presented in [3] in a non-trivial way in order to decide entailment under this semantic model. We also fixed in our implementation some subtle issues we discovered in the algorithm from [3]. Although SeLoger can decide entailment under intuitionistic semantics, since our target semantic model is the one presented in [1], we do not compare SeLoger to SLAD. We do not expect major differences to arise when comparing SLAD to SeLoger on the intersection of the logical languages supported by the tools.

2 Separation Logic

SeLoger decides satisfiability and entailment in the fragment of separation logic introduced in [1]. The syntax of the assertion language of this fragment is given by the following grammar, where x and y range over an infinite set of *variables*:

$$\begin{aligned}
\phi &::= \top \mid \bot \mid x = y \mid x \neq y \mid \phi \wedge \phi && (\textit{pure formulas})\\
\sigma &::= \mathsf{emp} \mid \mathsf{true} \mid \mathsf{pt}(x, y) \mid \mathsf{ls}(x, y) \mid \sigma * \sigma && (\textit{spatial formulas})\\
\alpha &::= (\phi; \sigma) && (\textit{assertions})
\end{aligned}$$

We call assertions of our assertion language *SL formulas*. For brevity, we only informally introduce the semantics of SL formulas. In [1], SL formulas are interpreted over memory models consisting of a *heap* and a *stack*. A heap is a function mapping a finite subset of an infinite domain of *heap cells* (usually the naturals) to heap cells. The elements of the domain of a heap are called *allocated heap cells*. A stack maps a finite subset of variables to heap cells, *i.e.*, it labels heap cells with variables. Pure formulas make Boolean judgements about stacks in the obvious way, *e.g.* a stack models $x = y$ if x and y are mapped to the same heap cell. Spatial formulas on the other hand make judgements about the shape of the heap. With emp a heap is required to have no allocated heap cells; true holds always; the *points-to relation* $\mathsf{pt}(x, y)$ requires that the heap consists of a single allocated cell labelled with x that maps to the heap cell labelled with y; and the *list relation* $\mathsf{ls}(x, y)$ requires that there be a chain of connected allocated heap cells starting in x and ending in y *with no repetitions*. Finally, $\sigma_1 * \sigma_2$ holds for a heap if the set of allocated heap cells can be partitioned into disjoint sets such that σ_1 holds on the first partition and σ_2 on the second. Last, given a memory model, an assertion $(\phi; \sigma)$ holds if the stack is a model of ϕ and the heap a model of σ.

Given SL formulas α, α', *satisfiability* asks whether there is a memory model in which α holds, and *entailment* is to decide whether α' holds in every memory

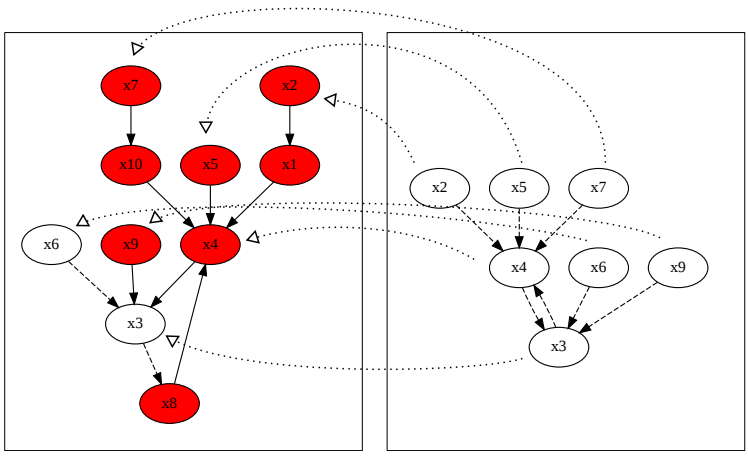


Fig. 1. Example of two SL graphs G_1 and G_2 and a graph homomorphism between them witnessing entailment between the corresponding SL formulas

model in which α holds, written $\alpha \models \alpha'$. We call α the *assumption* and α' the *goal* of the entailment.

3 A Sketch of the Graph-Based Entailment Algorithm

The key idea of the algorithm presented in [3] is to represent SL formulas as directed labelled coloured graphs (SL graphs). Entailment can then be decided by checking whether a canonical mapping from the set of nodes of the graph representing the goal to the set of nodes representing the assumption fulfils certain *homomorphism conditions*.

For brevity, instead of providing formal definitions, let us illustrate the representation of SL formulas as SL graphs and a homomorphism witnessing entailment between the formulas with the help of an example which can be found in Figure 1. The graph G_1 in the left-hand side box represents the SL formula $\alpha_1 = (\top; \sigma_1)$ and the graph G_2 in the right-hand side box the SL formula $\alpha_2 = (\top; \sigma_2)$, where

$$\sigma_1 = \mathsf{pt}(x_7, x_{10}) * \mathsf{pt}(x_2, x_1) * \mathsf{pt}(x_{10}, x_4) * \mathsf{pt}(x_5, x_4) * \mathsf{pt}(x_1, x_4) * \mathsf{pt}(x_4, x_3) * \\ * \mathsf{pt}(x_9, x_3) * \mathsf{pt}(x_8, x_4) * \mathsf{ls}(x_6, x_3) * \mathsf{ls}(x_3, x_8); \text{ and}$$
$$\sigma_2 = \mathsf{ls}(x_2, x_4) * \mathsf{ls}(x_5, x_4) * \mathsf{ls}(x_7, x_4) * \mathsf{ls}(x_4, x_3) * \mathsf{ls}(x_3, x_4) * \mathsf{ls}(x_6, x_3) * \mathsf{ls}(x_9, x_3).$$

Both SL formulas belong to an actual entailment instance found in the benchmark suite used in this paper and have been generated by SeLoger using GraphViz. The points-to relation is represented by solid arrows and the list relations by dashed arrows, nodes of the graphs correspond to equivalence classes of variables (here, each equivalence class is a singleton). The canonical mapping

h is represented by dotted arrows and witnesses that each list edge in G_2 has a corresponding path in G_1. For example, there is a path from the node labelled with x_3 to the node labelled with x_4 in G_1, which is required by the list edge from x_3 to x_4 in G_2. Furthermore, no edge in G_1 occurs along two paths that are induced by two different list edges of G_2 under h, and all edges in G_1 occur along a path that is induced by a list edge of G_2 under h. Together with some further technical conditions, we can show that h is a homomorphism and that consequently α_1 entails α_2.

In general, the SL graphs corresponding to SL formulas do not exhibit such a nice structure as the ones presented in Figure 1. However, it is shown in [3] that any SL formula α is equivalent to an SL formula α' whose corresponding SL graph G enjoys some nice structural properties, *e.g.* that between any two nodes there is at most one loop-free path. In [3], a saturation procedure (there called reduction procedure) is presented that given α computes such a graph G if α is satisfiable, and indicates that α is unsatisfiable otherwise. In summary, the decision procedure for an entailment $\alpha_1 \models \alpha_2$ presented in [3] can be broken into three parts:

(i) Construction of the SL graphs G_1 and G_2 representing α_1 and α_2
(ii) Saturation of α_1 and α_2 (which gives a satisfiability test as a byproduct)
(iii) Checking whether the canonical mapping from the nodes of G_2 to the nodes of G_1 is a homomorphism

4 Experimental Evaluation

We have tested SeLoger against the tool SLP, rev. 13591, by Navarro Peréz and Rybalchenko [5] on the benchmark suite[1] used in the same paper and one class of benchmarks generated by us. In [5], the authors compare SLP to the reasoners used in jStar and Smallfoot. Since SLP significantly outperforms both jStar and Smallfoot on essentially all test cases, we decided to only benchmark SeLoger against SLP.

The benchmarks suite in [5] consists of three classes of benchmarks called "spaguetti", "bolognesa" and "clones". The class "bolognesa" consists of 11 files, each containing 1000 entailment checks of the form $\alpha \models \alpha'$. Both α and α' are SL formulas which are generated at random according to some rules specified in [5]. Initially, both SL formulas range over ten variables and this number is increased in each file by one such that the last "bolognesa" file contains 1000 entailment checks over formulas with 20 variables. Similarly, the "spaguetti" class contains 11 files with 1000 entailment checks of the form $\alpha \models \bot$, where α is generated at random and the number of variables used in α increases by one starting from 10. In both classes, the random instances are chosen such that roughly 50% of the entailments are valid. Finally, the "clones" class contains real-world

[1] The benchmark suite can be downloaded at
`http://navarroj.com/research/tools/slp-benchmarks.tgz`

Table 1. Comparison of SLP and SeLoger on the benchmark set used in [5] and an additional class ("awkward"). All times are in ms.

B.mark	SLP	SeLoger	B.mark.	SLP	SeLoger	B.mark	SLP	SeLoger	B.mark	SLP	SeLoger
bo-10	1410	291	sp-10	1240	255	cl-01	65	14	aw-01	23	1
bo-11	1781	341	sp-11	2214	297	cl-02	67	20	aw-02	25	2
bo-12	2421	439	sp-12	8181	348	cl-03	82	26	aw-03	28	2
bo-13	11.9k	442	sp-13	15.6k	391	cl-04	93	34	aw-04	33	3
bo-14	5862	467	sp-14	15.2k	408	cl-05	117	44	aw-05	43	3
bo-15	3937	495	sp-15	18.6k	438	cl-06	147	52	aw-06	64	4
bo-16	7156	546	sp-16	3503	442	cl-07	207	62	aw-07	127	5
bo-17	14.2k	571	sp-17	94.2k	517	cl-08	364	72	aw-08	345	6
bo-18	20.8k	642	sp-18	5129	525	cl-09	826	84	aw-09	1157	7
bo-19	40.7k	705	sp-19	27.2k	549	cl-10	2466	95	aw-10	4492	8
bo-20	27.0k	752	sp-20	70.7k	595	cl-11	8794	105	aw-11	18.4k	10
						cl-12	34.2k	118	aw-12	76.2k	11
						cl-13	139.8k	130			

entailments. It consists of 13 files[2], each containing 209 entailments that were extracted from verification conditions generated by Smallfoot when run on the examples shipped with the tool. Some of the entailments require an enriched syntax since they include arbitrary data fields. The algorithm presented in [3] can, however, be straight-forwardly generalised to also allow for data fields as required by the benchmarks. Since the verification conditions are of a rather simple nature, in order to increase the complexity the "clones" class incrementally adds copies of the entailments to each entailment such that in the last benchmark file, each entailment consists of 13 copies of the original entailment. Last, we generated a benchmark class called "awkward", where the n-th instance consists of a *single* entailment of the form $*_{1\leq i\leq n}\mathsf{ls}(x_i, y_i) * \mathsf{ls}(x_i, z_i) * \mathsf{ls}(y_i, w_i) * \mathsf{ls}(z_i, w_i) \models *_{1\leq i\leq n}\mathsf{ls}(x_i, y_i) * \mathsf{ls}(y_i, x_i) * \mathsf{ls}(x_i, z_i) * \mathsf{ls}(z_i, x_i)$.

SeLoger is written in F# and, according to [5], SLP is implemented in Prolog and was provided to us as a binary file. We ran the SeLoger benchmarks on a Samsung Series 9 ultrabook with an Intel® Core™ i5-2467M 1.60 GHz processor with 4 GB DDR3 1066 MHz under Windows® 7 Home Premium (64-bit) and the SLP benchmarks on the same machine under Ubuntu Linux 12.04.1.

The results of the benchmarks are shown in Table 1 and illustrated in Figure 2. In Table 1, each column contains the average running time over ten runs. For SeLoger, the average coefficient of variation encountered was 0.05 with a standard deviation of 0.05, and for SLP the average coefficient of variation was 0.04 with a standard deviation of 0.07. We observe that SeLoger finishes on all benchmarks in less than 800ms, that it is up to 1075 times faster on the benchmarks from [5], and that the running time encountered in praxis appears almost linear, while it grows exponentially for SLP. We created the "awkward" benchmarks with the intention of exaggerating this difference. Also note that SeLoger behaves in general more robustly in the sense that the running times monotonically increase with the complexity of the benchmarks.

[2] In [5], the "clones" class only consists of eight files, however for better comparison we generated the additional five files using the benchmark generator used in [5].

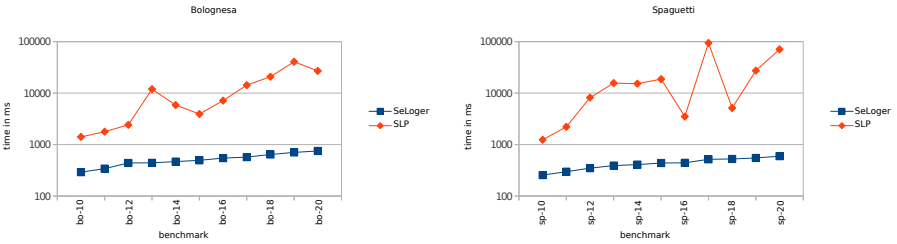


Fig. 2. Graphical illustration of some data from Table 1 on a logarithmic scale

5 Conclusion

In this paper, we introduced the tool SeLoger which implements and extends the entailment algorithm for the fragment of separation logic with pointers and linked lists described in [3]. We compared our tool to the tool SLP by Navarro Peréz and Rybalchenko [5]. Our benchmarks show that SeLoger outperforms SLP on all benchmarks considered and is often orders of magnitudes faster.

Together with other tools such as SLAD [2] that are based on the graph-based approach to entailment checking from [3], this suggests that this approach not only yields new complexity results, but also delivers practically-usable and high-performance algorithms. We are confident that SeLoger can serve as a basis for future work on graph-based algorithms and decision procedures for even richer fragments of separation logic and will find its way into future program verifiers.

Acknowledgement. We would like to thank Juan Antonio Navarro Peréz for making SLP available to us.

References

1. Berdine, J., Calcagno, C., O'Hearn, P.W.: A decidable fragment of separation logic. In: Lodaya, K., Mahajan, M. (eds.) FSTTCS 2004. LNCS, vol. 3328, pp. 97–109. Springer, Heidelberg (2004)
2. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Accurate invariant checking for programs manipulating lists and arrays with infinite data. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 167–182. Springer, Heidelberg (2012)
3. Cook, B., Haase, C., Ouaknine, J., Parkinson, M., Worrell, J.: Tractable reasoning in a fragment of separation logic. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 235–249. Springer, Heidelberg (2011)
4. Ishtiaq, S., O'Hearn, P.: BI as an assertion language for mutable data structures. In: Proceedings of POPL 2001, pp. 14–26. ACM (2001)
5. Peréz, J.A.N., Rybalchenko, A.: Separation logic + Superposition calculus = Heap theorem prover. In: Proceedings of PLDI 2011, San Jose, CA, USA. ACM Press (2011)
6. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: Proceedings of LICS 2002. IEEE Computer Society (2002)

Validating Library Usage Interactively*

William R. Harris, Guoliang Jin, Shan Lu, and Somesh Jha

University of Wisconsin, Madison, WI, USA
{wrharris,aliang,shanlu,jha}@cs.wisc.edu

Abstract. Programmers who develop large, mature applications often want to optimize the performance of their program without changing its semantics. They often do so by changing how their program invokes a library function or a function implemented in another module of the program. Unfortunately, once a programmer makes such an optimization, it is difficult for him to validate that the optimization does not change the semantics of the original program, because the original and optimized programs are equivalent only due to subtle, implicit assumptions about library functions called by the programs.

In this work, we present an interactive program analysis that a programmer can apply to validate that his optimization does not change his program's semantics. Our analysis casts the problem of validating an optimization as an abductive inference problem in the context of checking program equivalence. Our analysis solves the abductive equivalence problem by interacting with the programmer so that the programmer implements a solver for a logical theory that models library functions invoked by the program. We have used our analysis to validate optimizations of real-world, mature applications: the Apache software suite, the Mozilla Suite, and the MySQL database.

Keywords: abductive reasoning, program equivalence.

1 Introduction

Application developers often modify a program to produce a new program that executes faster than, but is semantically equivalent to, the original program. After a developer modifies his program, he can determine with high confidence whether the modified program executes faster than the original program by measuring the performance of the original and modified program on a set of performance benchmarks. Unfortunately, it is significantly harder for the developer to determine that the modified program is semantically equivalent to the original program.

* Supported, in part, by DARPA and AFRL under contract FA8650-10-C-7088. The views, opinions, and/or findings contained herein are those of the authors and should not be interpreted as representing the official views or policies, either expressed or implied, of DARPA or the Department of Defense. Also supported by NSF under grants CCF-1054616, CCF-1217582, and by a Clare Boothe Luce faculty fellowship.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 796–812, 2013.

Much previous work in developing correct compiler optimizations has focused on developing fully-automatic analyses that determine if two programs are equivalent [23,25]. Unfortunately, such analyses usually require that the two programs call the same procedures with the same argument values. However, many practical optimizations modify a program to call a library function on different values, or call a different library function entirely. Such analyses cannot prove that such an optimization preserves the semantics of a program. Other analyses attempt to determine if two programs are equivalent by analyzing the programs interprocedurally [10,17]. Unfortunately, many practical optimizations modify calls to complex, heavily optimized library functions. Such functions may be difficult to analyze, or their source code may be unavailable.

In this work, we propose a new interactive analysis for determining under what conditions two programs are equivalent. Unlike previous work, the analysis that we propose is not fully automatic. Instead, the analysis takes as input from a programmer an original program and an optimized program, and suggests a candidate specification of the functions defined in libraries and other program modules (in this paper, we refer to all such functions as "library" functions for simplicity) called by the program that implies that the original and optimized programs are equivalent. The programmer either validates or refutes the candidate specification, and the analysis uses this validation or refutation to iteratively suggest a new sufficient specification, until the analysis finds a sufficient specification that is validated by the programmer. If a programmer accepts an invalid specification of library functions, then the analysis may incorrectly determine that the programs are equivalent. However, even if a programmer accepts an invalid specification, the analysis still generates an explicit representation of the key assumptions made by the programmer to justify the optimization. The assumptions can potentially be validated by other programmers or known techniques for verifying safety properties of programs [4,6,13].

There are two key challenges to developing an interactive equivalence checker. The first key challenge is to develop a checker that can construct candidate specifications about functions whose implementations may not be available, or that manipulate complex abstract datatypes, such as strings, that are difficult to reason about symbolically. The equivalence checker must find specifications describing such functions that it can soundly determine to be consistent and sufficient to prove that the original program is equivalent to the optimized program.

The second key challenge is to develop an interactive checker that queries its user with simple, non-redundant candidate specifications about the library functions that a program calls. To prove equivalence between an original and optimized program, the interactive checker must work with the user to construct a simulation relation from the state space of the original program to the state space of the optimized program. However, the checker should largely hide from the user the complexity of constructing such a simulation relation, so that the user must only ever make simple Boolean decisions determining the validity of a candidate specification of library functions.

```
string srch_strm(string s) {
L0: string sb := "";
L1: while(!find(sb, s)) {
      char c := get();
      sb := append(sb, c);
    }
L2: return sb;
}
```

```
string srch_strm'(string s) {
L0':  str sb := "";
      int pos := - len(s);
L1':  while(pos < 0
            || !find(sub(sb, pos), s)) {
       char c := get();
       sb := append(sb, c);
       pos := len(sb) - len(s);
     }
L2': return sb;
}
```

Fig. 1. An example original program `srch_strm` and its optimization `srch_strm'`, simplifications of the original and optimized programs submitted in Apache Bug #34464.

Our key insight to address the above challenges is to design the checker so that it treats the user as a solver for a theory that describes the library functions. The equivalence checker reduces the problem of checking equivalence to proving the validity of a set of formulas, using known techniques for checking equivalence [23]. To prove validity of the required formulas, the equivalence checker applies an *abductive theorem prover*, which generates an assumption over the library functions, restricted to logical combinations of equalities, that are sufficient for each formula to be valid. To generate such an assumption, the theorem prover uses an *optimistic solver* for the theory of program libraries. If the optimistic solver finds consistent assumptions sufficient to prove validity, then the equivalence checker presents the assumptions for the user to validate or refute. In other words, the optimistic solver interacts with the user to implement a "guess-and-check" solver for the theory of the libraries.

The rest of this paper is organized as follows: §2 illustrates our equivalence problem and analysis on a function and its optimization submitted in a bug report for the Apache Ant build tool. §3 presents our abductive equivalence problem and analysis in detail. §4 presents an experimental evaluation of our analysis on a set of benchmarks taken from bug reports to fix performance issues in applications. §5 discusses related work.

2 Overview

In this section, we motivate the abductive equivalence problem and algorithm introduced in this paper using an optimization submitted in a bug report for the Apache Ant build tool [3]. Fig. 1 contains a program function `srch_strm` and an optimization of the function `srch_strm'` submitted in Apache Bug Report #34464 [2]. The actual original and optimized programs submitted in Bug Report #34464 use additional variables and control structure, and were written in Java, but have been simplified to `srch_strm` and `srch_strm'` in Fig. 1 to simplify the discussion. However, we have implemented our algorithm as a tool

that checks the equivalence of the actual programs (we translated the programs to C++ by hand so that we could apply our checker, which uses the LLVM compiler framework [19]).

`srch_strm` and `srch_strm'` implement an equivalent search for a substring in an input stream. Both functions take a string `s`, and read characters from a stream until they have read a string that contains `s`. `srch_strm` implements the search by constructing an empty string `sb` and iteratively checking if `sb` contains `s` as a subsequence. In each iteration, `srch_strm` calls `find(sb, s)`, which returns `True` if and only `sb` constrains `s` as a subsequence. If `find(sb, s)` = `True`, then `srch_strm` returns `sb`. Otherwise, `srch_strm` gets a character `c` from the stream, appends `c` to `sb`, and iterates again.

An Apache developer observed that while `srch_strm` outputs the correct value for each input string, it is inefficient due to how it uses the string library functions `find` and `append`. In Apache Bug Report #34464, the developer submitted a patch to `srch_strm`, called `srch_strm'`, that is functionally equivalent to `srch_strm`, but which the developer measured to be more efficient than `srch_strm`. `srch_strm'` is structured similarly to `srch_strm`, but executes more efficiently by only searching for `s` in a sufficiently long suffix of `sb`. In particular, `srch_strm'` maintains an integer variable `pos` that stores the position in `sb` from which `srch_strm'` searches for `s`. In each iteration, `srch_strm'` constructs the substring of `sb` starting at `pos`, `sub(sb, pos)`, tries to find `s` in `sub(sb, pos)`, and if it fails, gets a new character `c` from the stream, appends `c` to `sb`, and iterates again.

An Apache developer submitted `srch_strm'` with an informal argument that it is semantically equivalent to `srch_strm`, but ideally, the developer would submit `srch_strm'` accompanied by a proof that could be checked automatically to determine that `srch_strm` is equivalent to `srch_strm'`. Existing analyses for constructing automatically-checkable proofs of equivalence construct a *simulation relation* from P to P', which shows that every execution of P corresponds to an execution of P' that returns the same value [20,23,25]. A simulation relation $\sim$ from P to P' is a binary relation from the states of P to the states of P' such that: (1) $\sim$ relates each initial state of P to the state in P' with equal values in each variable; (2) $\sim$ relates each return state of P to a state of P' with the same return value; and (3) if $\sim$ relates a state q_0 of P to a state q_0' of P' and q_0 transitions to a state q_1 of P, then q_0' transitions, possibly over multiple steps, to some state q_1' of P' such that $\sim$ relates q_1 to q_1'.

One simulation relation from `srch_strm` to `srch_strm'`, under an intuitive semantics of the library functions `append`, `find`, `sub`, and `len`, is:

$$(L_0, L_0') : s = s' \tag{1}$$

$$(L_1, L_1') : s = s' \wedge sb = sb' \tag{2}$$

$$(L_2, L_2') : sb = sb' \tag{3}$$

The simulation relation $\sim_{\text{ex}}$ of Formulas (1)–(3) is represented as a map from a pair of program labels to a formula that describes pairs of program states. A state q of `srch_strm` at label `L` is related to a state q' of `srch_strm'` at label `L'`

if the values of the variables in q and q' satisfy the formula mapped from (L, L') (where the variables of q' are primed). $\sim_{\text{ex}}$ relates all initial states of `srch_strm` to initial states of `srch_strm'` with equal values for `s` (Formula (1)), all return states of `srch_strm` to return states of `srch_strm'` that return the same value (Formula (3)), and states at the loop head of `srch_strm` to states at the loop head of `srch_strm'` with equal values for `s` and `sb` (Formula (2)).

$\sim_{\text{ex}}$ is a straightforward instance of the definition of a simulation relation for `srch_strm` and `srch_strm'`. However, the fact that $\sim_{\text{ex}}$ satisfies condition (2) of a simulation relies on the semantics of the library functions `append`, `find`, and `sub` called by `srch_strm` and `srch_strm'`. Unfortunately, in practice it is difficult to automatically infer accurate specifications for library functions, as such functions may be unavailable or difficult to analyze.

Fortunately, while a programmer may not be able to give a complete formal specification of a library, they often understand a weaker, partial specification that implies the equivalence of a particular optimization. For example, `srch_strm` and `srch_strm'` are equivalent under the assumptions that (1) if the length of string sb is less than the length of string s, then $\texttt{find}(sb, s) = \mathsf{False}$ and (2) if $\texttt{find}(\texttt{sb}, \texttt{s}) = \mathsf{False}$, then `s` is a subsequence of the concatenation of `s` with a character `c` if and only if `s` is a subsequence of the suffix of `s` and `c` of length equal to the length of `s`:

$$\forall sb, s.\ \texttt{len}(sb) - \texttt{len}(s) < 0 \implies \neg\texttt{find}(sb, s) \tag{4}$$

$$\forall sb, s, c.\ \neg\texttt{find}(sb, s) \implies (\texttt{find}(\texttt{append}(sb, c), s) \iff \texttt{find}(\texttt{sub}(sb, \texttt{len}(sb) - \texttt{len}(s)), s)) \tag{5}$$

Based on the insight that programmers can often reliably validate partial specifications of libraries, in this work we introduce the *abductive equivalence* problem AEQ (§3.2). An abductive equivalence problem is defined by an original program P, optimized program P', and an *oracle*, which models a programmer, that takes a formula φ describing the library functions called by P and P' and accepts φ if the oracle's model of the library functions satisfies φ. A solution to the problem is a simulation relation from P to P' under the oracle's model of the library functions.

We present a sound algorithm for AEQ, called ChkAEQ (§3.3), that extends existing algorithms for checking program equivalence [23]. Like algorithms for checking equivalence, ChkAEQ first asserts that the return values of input programs P and P' are equal, and then reasons backwards over the executions of P and P' to construct a simulation relation from P to P' represented as a map from pairs of program control labels to formulas in a logic that describes the states of the program. The key feature of ChkAEQ is that as it constructs a relation $\sim$ from the states of P to P', it applies an *abductive theorem prover* to construct a condition on the library functions (i.e., a *library condition*) that implies that $\sim$ is a simulation relation. If ChkAEQ finds a simulation relation and sufficient library condition, it queries the input oracle on the library condition to determine if the oracle's model satisfies the library condition. If the library oracle validates the condition, then ChkAEQ returns the simulation relation. Otherwise, if the oracle

$$
\begin{array}{lll}
\mathsf{block} := \mathtt{L} : \mathsf{instr}; \mathsf{term} & & \mathtt{L} \in \mathsf{Labels} \\
\mathsf{instr} := \mathtt{x}_0 := \mathtt{f}(\mathtt{x}_1, \ldots, \mathtt{x}_n) \mid \mathtt{x}_0 := \mathtt{g}(\mathtt{x}_1, \ldots, \mathtt{x}_n) & & \{\mathtt{x}_i\}_i \subseteq \mathsf{Vars}; \mathtt{f} \in \mathsf{Ops}; \mathtt{g} \in \mathsf{LibOps} \\
\mathsf{term} := \mathtt{return\ x} \mid \mathtt{br\ x\ ?\ L_t\ :\ L_f} & & \mathtt{L_t}, \mathtt{L_f} \in \mathsf{Labels}; \mathtt{x} \in \mathsf{Vars}
\end{array}
$$

Fig. 2. Syntax of the programming language Imp, described in §3.1. An Imp program is a set of blocks.

refutes the condition, then ChkAEQ uses the refutation to continue to search for a simulation relation.

For `srch_strm` and `srch_strm'` (Fig. 1), ChkAEQ could infer that if each string s can be found in each string sb (i.e., $\forall sb, s.\ \mathtt{find}(sb, s)$ (6) is valid), then $\sim_{\text{ex}}$ is a simulation relation. However, a programmer serving as a library oracle would refute Formula (6). ChkAEQ would then use the refutation to search for a condition consistent with the negation of Formula (6), and would eventually find the library conditions of Formulas (4) and (5).

3 Abductive Equivalence

In this section, we formally define the abductive equivalence problem and algorithm. In §3.1, we define the syntax and semantics of a simple imperative language Imp. In §3.2, we define the abductive equivalence problem for Imp programs. In §3.3, we present a sound algorithm for solving the abductive equivalence problem.

3.1 IMP: A Simple Imperative Language

IMP Syntax. An Imp program updates its state by executing a sequence of program and library operations. The Imp language (Fig. 2) is defined over a set of variable symbols Vars, a set of control labels Labels, a set of language function symbols Ops, and a set of library function symbols LibOps, where Vars, Labels, Ops, and LibOps are mutually disjoint. Labels contains a label `RET` that does not label any block of an Imp program (`RET` is used to define the semantics of a return instruction; see "IMP Semantics").

An Imp program is a set of basic blocks. Each Imp program P contains one initial block labeled with a $\iota(P) \in \mathsf{Labels}$. Each basic block block is a label followed by an instruction and block terminator. An instruction is an assignment of either a language operation or a library operation. A block terminator is either a return instruction or a conditional branch.

IMP Semantics. The operational semantics of Imp (Fig. 3) defines how a basic block transforms a given state under a given model of library operations. An Imp *state* is a label paired with a *valuation*, which is a map from each program variable to an integer value: $\mathsf{Valuations} = \mathsf{Vars} \to \mathbb{Z}$ and

$$\sigma_b^m[\![\texttt{L} : \mathsf{instr}; \mathsf{term}]\!](L_0, V) \equiv \textbf{if } L_0 = L \textbf{ then } \sigma_t[\![\mathsf{term}]\!](\sigma_i^m[\![\mathsf{instr}]\!](V)) \textbf{ else } \bot \quad (6)$$

$$\sigma_i^m[\![\texttt{x}_0 := \texttt{f}(\texttt{x}_1)]\!](V) \equiv V[x_0 \mapsto \mathsf{langmodel}(\texttt{f})(V(x_1), \ldots, V(x_n)))] \quad (7)$$

$$\sigma_i^m[\![\texttt{x}_0 := \texttt{g}(\texttt{x}_1)]\!](V) \equiv V[x_0 \mapsto m(\texttt{g})(V(x_1), \ldots, V(x_n))] \quad (8)$$

$$\sigma_t[\![\texttt{return x}]\!](V) \equiv (\texttt{RET}, V[\mathsf{rv} \mapsto V(x)]) \quad (9)$$

$$\sigma_t[\![\texttt{br x ? L}_\texttt{t} \texttt{ : L}_\texttt{f}]\!](V) \equiv ((\textbf{if } V(x) \neq 0 \textbf{ then } L_t \textbf{ else } L_f), V) \quad (10)$$

Fig. 3. Operational semantics of IMP. σ_b^m is the operational semantics of a block and σ_i^m is the operational semantics of an instruction under library model m. σ_t is the operational semantics of a block terminator. In σ_b^m, $\bot$ denotes the undefined value. In σ_i^m and σ_t, for $a, b \in \mathbb{Z}$, $V[a \mapsto b]$ maps a to b, and maps $c \neq a$ to $V(c)$. In σ_t, $\mathsf{rv} \in \mathsf{Vars}$ stores the return value of the program.

$\mathsf{States} = \mathsf{Labels} \times \mathsf{Valuations}$. A *library model* $m : \mathsf{LibOps} \to (\mathbb{Z}^* \to \mathbb{Z})$ maps each library function to a function from a vector of integers to an integer.

The semantic function of a block σ_b^m (Fig. 3, Eqn. (6)) defines how a basic block transforms a state under m. The semantic function of an instruction σ_i^m (Fig. 3, Eqns. (7) and (8)) defines how an instruction updates a valuation. If an instruction assigns the result of a language operation, then the value of the operation is defined by a fixed *language model* $\mathsf{langmodel} : \mathsf{Ops} \to (\mathbb{Z}^* \to \mathbb{Z})$ (Fig. 3, Eqn. (7)). If an instruction assigns the result of a library operation, then the value of the operations is defined by m (Fig. 3, Eqn. (8)). The semantic function of a block terminator σ_t defines how a block terminator transforms a state (Fig. 3, Eqns. (9) and (10)).

3.2 The Abductive Equivalence Problem

To define the abductive equivalence problem, we adopt the definition of a simulation relation [20], which has been used to define the classical equivalence problem [23,25]. A simulation relation from an IMP program P to an IMP program P' is a relation from states of P to states in P' that implies that if from inputs I, P has an execution that returns value v, then from I, P' also has an execution that returns v.

Defn. 1 *Let* IMP *program* P *be* compatible *with* IMP *program* P' *if every variable of* P *corresponds to a variable of* P'*:* $\mathsf{Vars}(P) \subseteq \mathsf{Vars}(P')$. *A* simulation relation *from* P *to a compatible program* P' *under library model* $m : \mathsf{LibOps} \to (\mathbb{Z}^* \to \mathbb{Z})$ *is a relation* $\sim \subseteq \mathsf{States} \times \mathsf{States}$ *from the states of* P *to the states of* P' *such that:*

1. Initial states*:* $\sim$ *relates each initial state of* P *to its analogous state in* P'*. For valuations* $V, V' \in \mathsf{Valuations}$*, if for each* $x \in \mathsf{Vars}(P)$*,* $V(x) = V'(x)$*, then* $\sim ((\iota(P), V), (\iota(P'), V'))$.
2. Return states*:* $\sim$ *relates each return state of* P *to a return state of* P' *that returns an equal value. For the distinguished variable* $\mathsf{rv} \in \mathsf{Vars}$ *holding the return value of the fucntion, and for each* $V \in \mathsf{Valuations}$*, there is some* $V' \in \mathsf{Valuations}$ *such that* $V(\mathsf{rv}) = V'(\mathsf{rv})$ *and* $\sim ((\texttt{RET}, V), (\texttt{RET}, V'))$.

Algorithm 1. An abductive equivalence algorithm ChkAEQ. Takes as input a $\mathcal{T}_{lib}$ oracle $\mathcal{O}_m$ and two IMP programs P and P', and constructs a solution to $\mathsf{AEQ}(m, P, P')$ (Defn. 2). $\mathsf{Interact}(A)$ returns a simulation relation under a library condition that implies A and is validated by $\mathcal{O}_m$. ChkAEQ is discussed in §3.3.

```
1: function ChkAEQ(O_m, P, P')
2:     function Interact(A)
3:         (C, ~) := SimRel(P, P', A)
4:         if O_m(C) then return ~
5:         else return Interact(A ∧ ¬C)
6:         end if
7:     end function
8:     return Interact(True)
9: end function
```

3. Consecution*: if* $\sim$ *relates a state* s_0 *of* P *to a state* s_0' *of* P' *and* s_0 *transitions to a state* s_1 *of* P*, then* s_0' *eventually transitions to a state* s_1' *of* P' *such that* $\sim$ *relates* s_1 *to* s_1'*. For program* P*, let the transition relation* $\rightarrow_P \subseteq \mathsf{States} \times \mathsf{States}$ *relate states connected by the transition function of* P*: for states* $s_0, s_1 \in \mathsf{States}$*,* $\rightarrow_P (s_0, s_1)$ *if and only if there is some block* $B \in P$ *such that* $s_1 = \sigma_b^m[\![B]\!](s_0)$*. Let* $\rightarrow_P^* \subseteq \mathsf{States} \times \mathsf{States}$ *be the reflexive transitive closure of* $\rightarrow_P$*. For* $s_0, s_0', s_1 \in \mathsf{States}$*, if* $\sim (s_0, s_0')$ *and* $\rightarrow_P (s_0, s_1)$*, then there is some* $s_1' \in \mathsf{States}$ *such that* $\rightarrow_{P'}^* (s_0', s_1')$ *and* $\sim (s_1, s_1')$*.*

The abductive equivalence problem for programs P and P' and library model m is to find a simulation relation from P to P' under m using P, P', and an *oracle* for m that answers Boolean queries about properties of m. Intuitively, the oracle formalizes the role of a programmer who can answer queries about the specification of a library. The oracle answers queries on properties expressed as formulas of a logical theory $\mathcal{T}_{lib}$ whose models describe IMP's library operations. For each library operation $\mathsf{g} \in \mathsf{LibOps}$, there is an uninterpreted function symbol g in $\mathcal{T}_{lib}$. The only predicate of $\mathcal{T}_{lib}$ is the equality relation $=$. The set of *library conditions* $\mathsf{Forms}(\mathcal{T}_{lib})$ are the first-order formulas of $\mathcal{T}_{lib}$, and for library condition $\varphi \in \mathsf{Forms}(\mathcal{T}_{lib})$, $m \models \varphi$ denotes that m is a model of φ. For library model m, the *oracle* $\mathcal{O}_m \subseteq \mathsf{Forms}(\mathcal{T}_{lib})$ *accepts* a $\mathcal{T}_{lib}$ formula if and only if m satisfies the formula: for $\varphi \in \mathsf{Forms}(\mathcal{T}_{lib})$, $\mathcal{O}_m(\varphi)$ if and only if $m \models \varphi$.

Defn. 2 *For library model* m *and* IMP *programs* P *and* P'*, the* abductive equivalence problem $\mathsf{AEQ}(m, P, P')$ *is, given* $\mathcal{O}_m$*,* P*, and* P'*, to find a simulation relation* $\sim$ *from* P *to* P' *under* m*.*

3.3 A Sound Algorithm for Abductive Equivalence

Interacting with an Oracle to Solve AEQ. In this section, we present an algorithm that soundly tries to solve the abductive equivalence problem (Defn. 2),

Algorithm 2. SimRel: takes an original program P, optimized program P', and library condition A, and constructs a library condition C such that $C \implies A$ and a simulation relation from P to P' under C. $\mathsf{sym}_{\mathtt{ret}}$, ConsLocRel, and ConsecSimRel are discussed in §3.3.

```
 1: function SimRel(P, P', A)
 2:     lr ← ConsLocRel(P, P')
 3:     function ConsecSimRel(C, sym, W)
 4:         if W = ∅ then return (C, sym)
 5:         else
 6:             (L, L') ← RemElt(W)
 7:             cs := ⋁{wp_P^P' ⟦(L, L'), (L1, L1')⟧(sym(L1, L1')) | (L1, L1') ∈ succ_lr(L, L')}
 8:             suff := ATP(sym(L, L') ⟹ cs)
 9:             if IsTConsistent(suff) then C := C ∧ suff
10:             else
11:                 sym(L, L') := cs
12:                 W := W ∪ preds_lr(L, L')
13:             end if
14:             return ConsecSimRel(C, sym, W)
15:         end if
16:     end function
17:     (A_cs, sym_cs) := ConsecSimRel(A, sym_ret, dom(lr) ∪ img(lr))
18:     if IsTValid(initeq ⟹ sym_cs(ι(P), ι(P'))) then return (A_cs, sym_cs)
19:     else abort
20:     end if
21: end function
```

called ChkAEQ (Alg. 1). ChkAEQ tries to solve an abductive equivalence problem $\mathsf{AEQ}(m, P, P')$ as follows. ChkAEQ first defines a function Interact (Alg. 1, lines [2]–[6]) that takes a library condition A, and constructs a simulation relation from P to P' under m if $m \models A$. When ChkAEQ is successful, it returns the simulation relation constructed by applying Interact to True (Alg. 1, line [8]). However, ChkAEQ may fail or not terminate.

Interact constructs a simulation relation by applying a function SimRel (Alg. 1, line [3]) that takes an original program P, optimized program P', and a library condition A, and constructs (1) a library condition C such that $C \implies A$ and (2) a simulation relation from P to P' under each library model that satisfies C (in which case, we say that $\sim$ is a simulation from P to P' *under* C). If $\mathcal{O}_m$ accepts C (Alg. 1, line [4]), then Interact returns $\sim$ (Alg. 1, line [4]). Otherwise, Interact calls itself with a stronger library condition that asserts that C is not valid (Alg. 1, line [5]).

Constructing a Simulation Relation. SimRel constructs a simulation relation represented as a symbolic state relation, which is a function from pairs of labels to a formula of a theory $\mathcal{T}$ for which each model defines a library model, a state of P, and a state of P'. The theory $\mathcal{T}_{\text{IMP}}$ describes states of fixed programs P and P'. For each $\mathtt{f} \in \mathsf{Ops}$, let there be a unary function f in the logical theory

$\mathcal{T}_{\text{IMP}}$. For each program variable $\mathsf{x} \in \mathsf{Vars}(P)$, let there be a $\mathcal{T}_{\text{IMP}}$ constant x, and for each $\mathsf{x} \in \mathsf{Vars}(P')$, let there be a $\mathcal{T}_{\text{IMP}}$ constant x' that does not correspond to any variable in $\mathsf{Vars}(P)$. Let the only predicate of $\mathcal{T}_{\text{IMP}}$ be the equality predicate, which $\mathcal{T}_{\text{IMP}}$ shares with $\mathcal{T}_{lib}$ (§3.2). Let the combination of $\mathcal{T}_{lib}$ and $\mathcal{T}_{\text{IMP}}$ be $\mathcal{T} = \mathcal{T}_{lib} + \mathcal{T}_{\text{IMP}}$ [24]. A symbolic state relation $\mathsf{sym} : (\mathsf{Labels} \times \mathsf{Labels}) \to \mathsf{Forms}(\mathcal{T})$ relates states $(L, v), (L', v') \in \mathsf{States}$ under library condition C if for each library model $m \models C$, $m \cup v \cup v' \models \mathsf{sym}(L, L')$.

SimRel constructs a library condition A_{cs} such that $A_{cs} \implies A$, and a symbolic state relation sym_{cs} that satisfies the consecution condition of a simulation relation (Defn. 1, item 3) under A_{cs}. SimRel defines a *label transition relation* $\mathsf{lr} \subseteq (\mathsf{Labels} \times \mathsf{Labels}) \times (\mathsf{Labels} \times \mathsf{Labels})$, which defines the domain of sym_{cs}, by applying a function ConsLocRel (Alg. 2, line [2]). ConsLocRel can be implemented using known techniques and heuristics from classical equivalence checking [23]. SimRel then defines a function ConsecSimRel (Alg. 2, lines [3]–[16]) that takes as input (1) a library condition C, (2) a symbolic state relation sym, and (3) a workset W of label pairs on which sym may not satisfy Defn. 1, item 3 under C, and constructs a library condition A_{cs} and a state relation sym_{cs} that satisfy the consecution condition, and such that A_{cs} implies C. To construct sym_{cs} and A_{cs}, SimRel applies ConsecSimRel to A, a simulation relation $\mathsf{sym}_{\mathtt{ret}}$, and all pairs of labels in the domain and image of lr (Alg. 2, line [17]). $\mathsf{sym}_{\mathtt{ret}}$ satisfies the return condition of a simulation relation, defined in Defn. 1, item 1.

SimRel then checks that sym_{cs} satisfies the condition for a simulation relation on initial blocks (Defn. 1, item 1) by checking that $\mathsf{initeq} \equiv \bigwedge_{x \in \mathsf{Vars}(P)} x = x'$ implies the relation of states at the initial blocks of P and P' (Alg. 2, line [18]). If so, then SimRel returns $(A_{cs}, \mathsf{sym}_{cs})$ as a simulation relation (Alg. 2, line [18]). Otherwise, SimRel fails (Alg. 2, line [19]).

ConsecSimRel (Alg. 2, lines [3]–[16]) first checks if its workset of labels W is empty (Alg. 2, line [4]), and if so, returns its input library condition C and input state relation sym (Alg. 2, line [4]). Otherwise, ConsecSimRel chooses a pair of a labels (L, L') from W (Alg. 2, line [6]) on which it will update sym. For $\varphi \in \mathsf{Forms}(\mathcal{T})$, let $\mathsf{wp}_P^{P'}[\![(L_0, L_0'), (L_1, L_1')]\!](\varphi)$ be the formula whose models define states that transition to states in φ over steps of execution in P from L_0 to L_1 and steps of execution in P' from L_0' to L_1' ($\mathsf{wp}_P^{P'}$ is defined from the semantics of IMP (Fig. 3) using well-known techniques [11]). ConsecSimRel constructs cs, the disjunction of the weakest precondition of each formula to which sym maps each successor of L and L' under lr (Alg. 2, line [7]). ConsecSimRel then tries to construct a library condition $\mathsf{suff} \in \mathsf{Forms}(\mathcal{T}_{lib})$ that is consistent and implies that $\mathsf{sym}(L, L')$ implies cs. To construct suff, ConsecSimRel applies an abductive theorem prover ATP (Alg. 2, line [8]; ATP is discussed in [12], App. A). If suff is consistent, then ConsecSimRel conjoins suff to C (Alg. 2, line [9]). Otherwise, ConsecSimRel updates sym to map L and L' to cs, and adds each predecessor of L and L' under lr to W (Alg. 2, lines [11]–[12]). ConsecSimRel then calls itself recursively on its updated library condition, symbolic state relation, and workset.

Correctness of ChkAEQ. AEQ is at least as hard as determining if two Imp programs are equivalent. Imp is Turing-complete, and thus AEQ is undecidable. However, ChkAEQ is sound, but not complete, for AEQ. ChkAEQ also does not pose redundant queries to its oracle. These claims are formalized in an extended version of this paper ([12], Sec. 3).

Suppose that programs P and P' are not equivalent under library model m. Then for AEQ(mP, P'), ChkAEQ either will not terminate, or will abort when it fails to find a simulation relation that relates the program points guessed by ConsLocRel. It would be interesting to extend ChkAEQ so that it simultaneously searches for sufficient library conditions under which programs are equivalent, or sufficient library conditions that prove that the programs are definitely *not* equivalent. If a programmer submits a patch that is not equivalent to their original program, then ChkAEQ extended in this way could explain to the programmer why their patch is incorrect.

4 Experiments

We carried out a set of experiments to determine if programmers can apply ChkAEQ (§3) to validate practical optimizations. The experiments were designed to answer the following questions:

1. Given a function from a real-world program and its optimization, can ChkAEQ quickly find a library condition that is sufficient to prove that the programs are equivalent?
2. Can ChkAEQ find library conditions that are small and easy for a programmer to validate?

To answer these questions, we implemented ChkAEQ as a tool, `chklibs`, and applied `chklibs` to a set of program functions and their optimizations. Each function was taken from a mature, heavily-used program, namely the Apache software suite, Mozilla Suite, or MySQL database. Each original program function was the subject of a bug report reporting that the function's behavior was correct, but that its performance was inefficient. Each corresponding optimized function was the patched, optimized function provided in the bug report. We interacted with `chklibs` to find library conditions that were sufficient to prove that the optimization was correct, and were valid according to our understanding of the libraries.

The results of the experiments indicate that ChkAEQ can be applied to validate practical optimizations. In particular:

1. `chklibs` quickly inferred library conditions that were sufficient to prove equivalence. `chklibs` usually found validated sufficient library conditions in less than a second, and always found validated sufficient library conditions in less than 30 seconds (see Table 1).
2. `chklibs` often inferred syntactically compact sufficient library conditions. `chklibs` usually needed to suggest less than 10 disjunctive clauses until it

Table 1. Experimental data from using `chklibs`. The data given for each benchmarks program includes the name and of the source program, the bug report that presented the optimization, the number of lines of code of the original and optimized program functions, and the number of lines output by `diff` on the original and optimized programs. The data measuring `chklibs`'s performance includes the time taken by `chklibs` to construct a simulation relation (in seconds), the number of clauses on which `chklibs` queried the user, the average size of (i.e, number of logical symbols in) the clause, and the average number of predicates in each clause.

Benchmark Data					`chklibs` Performance			
Program	Bug	LoC			Analysis	Num.	Avg. Query Size	
Name	ID	Org.	Opt.	Diff.	Time (s)	Clauses	Clause Size	Num. Preds
Apache	19101	27	28	5	0.325	2	9.5	6.0
	34464	23	20	34	18.188	5	9.0	6.4
	44408	51	52	6	0.050	1	8.0	6.0
	45464	569	570	6	0.165	1	8.0	6.0
	48778	30	28	16	0.534	7	11.4	6.0
Mozilla	103330	217	216	5	0.064	2	18.0	6.0
	124686	198	198	4	0.096	1	278.0	6.0
	267506	182	184	9	0.507	5	8.0	6.0
	409961	54	57	12	0.795	3	47.0	6.0
MySQL	38769	223	227	4	0.169	2	11.0	6.0
	38824	346	321	18	29.894	13	179.2	6.0

suggested a sufficient set of clauses. The clauses always contained less than 10 predicates (and usually contained less than 5 predicates), and with some exceptions discussed below, were small enough that a programmer should be able to reason about their validity.

In §4.1, we describe in detail our procedure for evaluating ChkAEQ. In §4.2, we present and analyze the results of applying ChkAEQ.

4.1 Experimental Procedure

Implementation. `chklibs` solves the abductive equivalence problem for the LLVM [19] intermediate language. To implement `chklibs`, we extended the operational semantics of Imp (§3.1) to an operational semantics for the LLVM intermediate language, which included describing various language features such as structs and pointers. Such an extension is standard, and we omit its details. `chklibs` is implemented in about 5,000 lines of OCaml code, and uses the Z3 theorem prover [7] to implement the abductive theorem prover ATP ([12], App. A). `chklibs` simplifies each query and presents the query to the user as a conjunction of disjunctive clauses. We discuss simplifications that `chklibs` applies to queries in an extended version of this paper ([12], App. B).

Evaluation. To evaluate `chklibs`, we used it to validate a set of optimizations submitted to improve the performance of real-world applications. In particular,

we collected a set of bug reports from the public bug databases of Apache software suite [1], Mozilla Suite [21], and MySQL database [22] that each reported a performance issue and included a patch to fix the issue. We compiled each original program and its patch to the LLVM intermediate language. If a program and patch were originally implemented in a language supported by LLVM, such as C or C++, then we compiled the programs by applying the appropriate LLVM compiler front-end (`clang` or `clang++`, respectively). Otherwise, we rewrote the program functions by hand in C source code and compiled the source to the LLVM intermediate language by applying `clang`.

We applied `chklibs` to each original and optimized program, and interacted with `chklibs` to find library conditions that `chklibs` determined to be sufficient, and which were valid under our understanding of the libraries and program functions described informally in the bug report. In other words, we served as the library oracle introduced in §3.3. For each benchmark, we observed whether or not `chklibs` found library conditions that we believed to be valid, measured the total time spent by `chklibs` to infer sufficient specifications, measured the number of queries issued by `chklibs`, and measured the size of each query.

4.2 Results and Analysis

Results. Table 1 contains the results of applying `chklibs`. Each row in Table 1 contains data for a benchmark original and optimized program. In particular, Table 1 contains the name of the program from which the benchmark was taken, the ID of the bug report in which the optimization was submitted, the number of lines of code of the original and optimized program functions, the number of lines output by `diff` on the original and optimized programs, the time spent by `chklibs` to construct a validated simulation relation (not including the time spent by us to validate or refute a query posed by `chklibs`), the number of clauses on which `chklibs` queried the user, and average size of (i.e., the number of all logical symbols in) the clauses, and the average number of predicates in each clause. The size of the clause is the number of logical symbols in the clause.

Analysis. The data presented in Table 1 indicates that `chklibs` can be applied to suggest sufficient library conditions for equivalence that can often be easily validated by a programmer. In benchmarks where `chklibs` took an unusually long time to find validated sufficient conditions (Apache Bug 34464 and MySQL Bug 38824), `chklibs` posed a proportionally large number of queries that we refuted. In benchmarks where `chklibs` queried the user on an unusually large set of clauses (Apache Bug 48778 and MySQL Bug 38824), the original and optimized programs called different library functions at a proportionately large set of callsites. In benchmarks where `chklibs` queried the user on unusually large formulas (Mozilla Bug 124686 and MySQL Bug 38824), the formulas typically were constructed from equality conditions over addresses in the program that seem unlikely to alias. We believe that the size of these queries could be reduced drastically by combining `chklibs` with a more sophisticated alias analysis, or

a programmer with a more detailed understanding of the calling conventions of the original and optimized function.

We have provided a website[1] that contains, for each benchmark, the list of all queries generated by `chklibs` that we answered. Apache Bug #19101 and Mozilla Bug #409961 illustrate a common kind of optimization in which a programmer performs an interprocedural version of a classic compiler optimization: in Apache Bug #19101 and Mozilla Bug #409961, the optimization is loop-invariant code motion. To validate the optimizations, `chklibs` generates queries that determine sufficient library conditions to support the optimization. For Apache Bug #19101, `chklibs` queries if a method called within a loop does not change the values stored at particular fields of the calling object. For Mozilla Bug #409961, `chklibs` must determine that the final values returned by the programs are equivalent, even though some intermediate values computed within a loop may be different as a result of the optimization.

Apache Bug #48778 illustrates that in practical optimizations, there may be multiple consistent conditions on library that are sufficient to support an optimization. In Apache Bug #48778, the values returned by library functions determine the values of control-flow guards. `chklibs` correctly determines that if the guard values are constant, then the programs are equivalent. We refuted the corresponding library condition, which caused `chklibs` to eventually find a valid library condition equating the return values of particular library functions.

MySQL Bug #38769 illustrates how `chklibs` can make explicit supporting conditions that may be non-obvious to the developers. MySQL Bug #38769 optimizes a loop over an array by replacing a constant loop bound in the original program with the result of a method call, which may be a lower value. `chklibs` first determines that if the result of the method call is equal to the replaced constant, then the programs are equal. We refuted this condition, at which point `chklibs` determined that if all entries after the bound returned by the call are null, then the programs are equivalent, and we accepted this condition. For the report in which the patch was submitted, a developer notes that this condition did not hold for older versions of MySQL.

5 Related Work

Previous work [15] identified *performance bugs*, i.e., functionally correct but inefficient code, as a serious problem in commonly-used applications. In that work, the authors studied a set of performance bugs for five software suites, namely Apache, Chrome, GCC, Mozilla, and MySQL, derived a set of rules for identifying performance bugs manually from performance-bug reports, and statically checked programs to find new performance bugs that satisfy the rules. We have presented a technique and tool that allows a developer who submits a patch of a performance bug to validate conditions under which the patch preserves functionality of the program. We have applied to the tool to patches of bugs identified in the previous work on performance bugs.

[1] `http://pages.cs.wisc.edu/~wrharris/chklibs/`

Much existing work has focused on determining the equivalence of programs. Translation validation [23,25] is the problem of determining if a source program is equivalent to an optimized program, and is often applied to validate the correctness of the phases of an optimizing compiler. Regression verification [10] determines if a program and a similar revision of the program are equivalent. Semantic differencing [14] summarizes the different behaviors of two programs. Symbolic execution has been applied to determine the equivalence of loop-free programs [5]. In this work, we address the problem of taking an original and optimized program and inferring conditions on the libraries invoked by the programs that are sufficient to prove that the programs are equivalent, and that are validated by an oracle who understands the libraries. To solve the problem, we have extended an existing analysis for checking the equivalence of imperative programs with loops [23] to use the results of an abductive theorem prover.

The SymDiff project [16,17,18] shares our goal of determining under what conditions two programs are correct. Existing work in SymDiff takes a concurrent program, constructs a sequential version of the program, and treats the sequential version as a reference implementation, searching the concurrent program only for bugs triggered by inputs that cause no error in the sequential program [16]. Existing work on conditional equivalence [17] takes an original and optimized program and infers sufficient conditions on the inputs of a program under which the original and optimized programs are equivalent, where the space of conditions forms a lattice. In contrast, our work interacts with a user to infer sufficient conditions on the libraries invoked by an original and optimized program, and represents conditions as logical formulas. Given that the spaces of conditions described in techniques based on conditional equivalence must form a lattice, it is not immediately clear how to extend such a technique to interact with a user who may refute an initial condition suggested by the technique.

Recent work has extended the problem of deciding if a program always satisfies an assertion to an abductive setting, in which the problem is to find assumptions on the state of a program that imply that the program satisfies an assertion, and are validated by an oracle that answers queries about program states [8]. That work presents an algorithm that constructs sufficient assumptions by finding a minimum satisfying assignment of variables in a given formula [9], universally quantifying all unassigned variables, and eliminating the quantified variables using symbolic reasoning. Our work extends a different traditional problem in program analysis, that of checking program equivalence, to an abductive setting. While work on abductive assertion checking assumes that the theories for describing states of a program support quantifier elimination (e.g., linear arithmetic), we consider inferring assumptions for theories that may describe arbitrary library functions. Accordingly, our analysis applies an abductive theorem prover that does not assume that the theories modeling the semantics of a program support quantifier elimination, and instead generates assumptions as Boolean combinations of equality predicates.

Our approach to the abductive equivalence problem collects information from a programmer about the libraries that a program uses by querying the

programmer for the validity of purely equational formulas over the library functions, and propagates the logical consequence of the equalities to the rest of the analysis. Our approach is inspired by equality propagation [24], which is a technique for combining solvers for theories whose only shared predicate is equality to solve a formula defined in the combination of the theories. Essentially, our approach uses the programmer as a theory solver for the theory modeling library functions.

References

1. Apache (January 2013), `http://apache.org`
2. Apache bug #34464 (January 2013), `http://issues.apache.org/bugzilla/show_bug.cgi?id=34464`
3. Apache Ant (May 2012), `http://ant.apache.org`
4. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
5. Clarke, E.M., Kroening, D., Yorav, K.: Behavioral consistency of C and Verilog programs using bounded model checking. In: DAC (2003)
6. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)
7. de Moura, L.M., Bjørner, N.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
8. Dillig, I., Dillig, T., Aiken, A.: Automated error diagnosis using abductive inference. In: PLDI (2012)
9. Dillig, I., Dillig, T., McMillan, K.L., Aiken, A.: Minimum satisfying assignments for SMT. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 394–409. Springer, Heidelberg (2012)
10. Godlin, B., Strichman, O.: Regression verification. In: DAC (2009)
11. Gries, D.: The Science of Programming. Springer (1981)
12. Harris, W.R., Jin, G., Lu, S., Jha, S.: Validating library usage interactively (January 2013), `http://pages.cs.wisc.edu/~wrharris/validating_library_usage.pdf`
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
14. Jackson, D., Ladd, D.A.: Semantic diff: A tool for summarizing the effects of modifications. In: ICSM (1994)
15. Jin, G., Song, L., Shi, X., Scherpelz, J., Lu, S.: Understanding and detecting real-world performance bugs. In: PLDI (2012)
16. Joshi, S., Lahiri, S.K., Lal, A.: Underspecified harnesses and interleaved bugs. In: POPL (2012)
17. Kawaguchi, M., Lahiri, S.K., Rebelo, H.: Conditional equivalence. Technical Report MSR-TR-2010-119, Microsoft Research (October 2010)
18. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SymDiff: A language-agnostic semantic diff tool for imperative programs. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 712–717. Springer, Heidelberg (2012)
19. The LLVM compiler infrastructure

20. Milner, R.: Communication and concurrency. PHI Series in computer science. Prentice Hall (1989)
21. Mozilla – home of the Mozilla Project (2011), http://www.mozilla.org/
22. MySQL: The world's most popular open source database (2012), http://www.mysql.com/
23. Necula, G.C.: Translation validation for an optimizing compiler. In: PLDI (2000)
24. Nelson, G., Oppen, D.C.: Simplification by cooperating decision procedures. ACM Trans. Program. Lang. Syst. 1(2), 245–257 (1979)
25. Pnueli, A., Siegel, M.D., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998)

Learning Universally Quantified Invariants of Linear Data Structures

Pranav Garg[1], Christof Löding[2], P. Madhusudan[1], and Daniel Neider[2]

[1] University of Illinois at Urbana-Champaign
[2] RWTH Aachen University

Abstract. We propose a new automaton model, called *quantified data automata* over words, that can model quantified invariants over linear data structures, and build poly-time active learning algorithms for them, where the learner is allowed to query the teacher with membership and equivalence queries. In order to express invariants in decidable logics, we invent a decidable subclass of QDAs, called elastic QDAs, and prove that every QDA has a unique minimally-over-approximating elastic QDA. We then give an application of these theoretically sound and efficient active learning algorithms in a passive learning framework and show that we can efficiently learn quantified linear data structure invariants from samples obtained from dynamic runs for a large class of programs.

1 Introduction

Synthesizing invariants for programs is one of the most challenging problems in verification today. In this paper, we are interested in using *learning* techniques to synthesize quantified data-structure invariants.

In an *active* black-box learning framework, we look upon the invariant as a set of configurations of the program, and allow the learner to query the teacher for membership and equivalence queries on this set. Furthermore, we fix a particular representation class for these sets, and demand that the learner learn the smallest (simplest) representation that describes the set. A learning algorithm that learns in time polynomial in the size of the simplest representation of the set is desirable. In *passive* black-box learning, the learner is given a sample of examples and counter-examples of configurations, and is asked to synthesize the simplest representation that includes the examples and excludes the counter-examples. In general, several active learning algorithms that work in polynomial time are known (e.g., learning regular languages represented as DFAs [1]) while passive polynomial-time learning is rare (e.g., conjunctive Boolean formulas can be learned but general Boolean formulas cannot be learned efficiently, automata cannot be learned passively efficiently) [2].

In this paper, we build active learning algorithms for *quantified logical formulas describing sets of linear data-structures*. Our aim is to build algorithms that can learn formulas of the kind "$\forall y_1, \ldots, y_k\ \varphi$", where φ is quantifier-free, and that captures properties of arrays and lists (the variables range over indices for arrays, and locations for lists, and the formula can refer to the data stored at

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 813–829, 2013.

these positions and compare them using arithmetic, etc.). Furthermore, we show that we can build learning algorithms that learn properties that are expressible in known decidable logics. We then employ the active learning algorithm in a *passive learning* setting where we show that by building an imprecise teacher that answers the questions of the active learner, we can build effective invariant generation algorithms that learn simply from a finite set of examples.

Active Learning of Quantified Properties Using QDAs: Our first technical contribution is a novel representation (normal form) for quantified properties of linear data-structures, called *quantified data automata* (QDA), and a polynomial-time active learning algorithm for QDAs.

We model linear data-structures as *data words*, where each position is decorated with a letter from a finite alphabet modeling the program's pointer variables that point to that cell in the list or index variables that index into the cell of the array, and with data modeling the data value stored in the cell, e.g., integers. Quantified data automata (QDA) are a new model of automata over data words that are powerful enough to express *universally* quantified properties of data words. A QDA accepts a data word provided it accepts *all possible* annotations of the data word with valuations of a (fixed) set of variables $Y = \{y_1, \ldots, y_k\}$; for each such annotation, the QDA reads the data word, records the data stored at the positions pointed to by Y, and finally checks these data values against a data formula determined by the final state reached. QDAs are very powerful in expressing typical invariants of programs manipulating lists and arrays, including invariants of a wide variety of searching and sorting algorithms, maintenance of lists and arrays using insertions/deletions, in-place manipulations that destructively update lists, etc.

We develop an efficient active learning algorithm for QDAs. By using a combination of *abstraction* over a set of data formulas and Angluin's learning algorithm for DFAs [1], we build a learning algorithm for QDAs. We first show that for any set of valuation words (data words with valuations for the variables Y), there is a *canonical* QDA. Using this result, we show that learning valuation words can be reduced to learning *formula words* (words with no data but paired with data formulas), which in turn can be achieved using Angluin-style learning of Moore machines. The number of queries the learner poses and the time it takes is bound polynomially in the size of the canonical QDA that is learned. Intuitively, given a set of pointers into linear data structures, there is an exponential number of ways to permute the pointers into these and the universally quantified variables; the learning algorithm allows us to search this space using only polynomial time in terms of the actual permutations that figure in the set of data words learned.

Elastic QDAs and a Unique Minimal Over-Approximation Theorem: The class of quantified properties that we learn in this paper (we can synthesize them from QDAs) is very powerful. Consequently, even if they are learnt in an invariant-learning application, we will be unable to *verify* automatically whether the learnt properties are adequate invariants for the program at hand. Even though SMT solvers support heuristics to deal with quantified theories (like e-matching), in our experiments, the verification conditions could not be handled

by such SMT solvers. The goal of this paper is hence to also offer mechanisms to *learn invariants that are amenable to decision procedures.*

The second technical contribution of this paper is to identify a subclass of QDAs (called elastic QDAs) and show two main results for them: (a) elastic QDAs can be converted to formulas of *decidable* logics, to the array property fragment when modeling arrays and the decidable STRAND fragment when modeling lists; (b) a surprising *unique minimal over-approximation theorem* that says that for every QDA, accepting say a language L of valuation-words, there is a *minimal* (with respect to inclusion) language of valuation-words $L' \supseteq L$ that is accepted by an elastic QDA.

The latter result allows us to learn QDAs and then apply the unique minimal over-approximation (which is effective) to compute the best over-approximation of it that can be expressed by elastic QDAs (which then yields decidable verification conditions). The result is proved by showing that there is a unique way to minimally morph a QDA to one that satisfies the elasticity restrictions. For the former, we identify a common property of the array property fragment and the syntactic decidable fragment of STRAND, called *elasticity* (following the general terminology in the literature on STRAND [3]). Intuitively, both the array property fragment and STRAND prohibit quantified cells to be tested to be bounded distance away (the array property fragment does this by disallowing arithmetic expressions over the quantified index variables [4] and the decidable fragment of STRAND disallows this by permitting only the use of $\rightarrow^*$ or $\rightarrow^+$ in order to compare quantified variables [3,5]). We finally identify a *structural restriction* of QDAs that permits only elastic properties to be stated.

Passive Learning of Quantified Properties: The active learning algorithm can itself be used in a verification framework, where the membership and equivalence queries are answered using under-approximate and deductive techniques (for instance, for iteratively increasing values of k, a teacher can answer membership questions based on bounded and reverse-bounded model-checking, and answer equivalence queries by checking if the invariant is adequate using a constraint solver). In this paper, we do not pursue an implementation of active learning as above, but instead build a passive learning algorithm that uses the active learning algorithm.

Our motivation for doing passive learning is that we believe (and we validate this belief using experiments) that in many problems, a lighter-weight passive-learning algorithm which learns from a few randomly-chosen small data-structures is sufficient to find the invariant. Note that passive learning algorithms, in general, often boil down to a guess-and-check algorithm of some kind, and often pay an exponential price in the property learned. Designing a passive learning algorithm using an active learning core allows us to build more interesting algorithms; in our algorithm, the inacurracy/guessing is confined to the way the teacher answers the learner's questions.

The passive learning algorithm works as follows. Assume that we have a finite set of configurations S, obtained from sampling the program (by perhaps just running the program on various random small inputs). We are required to learn

the simplest representation that captures the set S (in the form of a QDA). We now use an active learning algorithm for QDAs; membership questions are answered with respect to the set S (note that this is imprecise, as an invariant I must include S but need not be precisely S). When asked an equivalence query with a set I, we check whether $S \subseteq I$; if yes, we can check if the invariant is adequate using a constraint solver and the program.

It turns out that this is a good way to build a passive learning algorithm. First, enumerating random small data-structures that get manifest at the header of a loop fixes for the most part the structure of the invariant, since the invariant is forced to be expressed as a QDA. Second, our active learning algorithm for QDAs promises never to ask long membership queries (queried words are guaranteed to be less than the diameter of the automaton), and often the teacher has the correct answers. Finally, note that the passive learning algorithm answers membership queries with respect to S; this is because we do not know the true invariant, and hence err on the side of keeping the invariant semantically small. This inaccuracy is common in most learning algorithms employed for verification (e.g, Boolean learning [6], compositional verification [7,8], etc). This inaccuracy could lead to a non-optimal QDA being learnt, and is precisely why our algorithm need not work in time polynomial in the simplest representation of the concept (though it is polynomial in the invariant it finally learns).

The proof of the efficacy of the passive learning algorithm rests in the experimental evaluation. We implement the passive learning algorithm (which in turn requires an implementation of the active learning algorithm). By building a teacher using dynamic test runs of the program and by pitting this teacher against the learner, we learn invariant QDAs, and then over-approximate them using elastic QDAs (EQDAs). These EQDAs are then transformed into formulas over decidable theories of arrays and lists. Using a wide variety of programs manipulating arrays and lists, ranging from several examples in the literature involving sorting algorithms, partitioning, merging lists, reversing lists, and programs from the Glib list library, programs from the Linux kernel, a device driver, and programs from a verified-for-security mobile application platform, we show that we can effectively learn adequate quantified invariants in these settings. In fact, since our technique is a black-box technique, we show that it can be used to infer pre-conditions/post-conditions for methods as well.

Related Work: For invariants expressing properties on the dynamic heap, *shape analysis* techniques are the most well known [9], where locations are classified/merged using *unary* predicates (some dictated by the program and some given as instrumentation predicates by the user), and abstractions summarize all nodes with the same predicates into a single node. The data automata that we build also express an infinite set of linear data structures, but do so using automata, and further allow n-ary quantified relations between data elements. In recent work, [10] describes an abstract domain for analyzing list manipulating programs, that can capture quantified properties about the structure and the data stored in lists. This domain can be instantiated with any numerical domain for the data constraints and a set of user-provided patterns for

capturing the structural constraints. However, providing these patterns for quantified invariants is in general a difficult task.

In recent years, techniques based on *Craig's interpolation* [11] have emerged as a new method for invariant synthesis. Interpolation techniques, which are inherently white-box, are known for several theories, including linear arithmetic, uninterpreted function theories, and even quantified properties over arrays and lists [12,13,14,15]. These methods use different heuristics like term abstraction [14], preferring smaller constants [12,13] and use of existential ghost variables [15] to ensure that the interpolant converges on an invariant from a *finite* set of spurious counter-examples. IC3 [16] is another white-box technique for generalizing inductive invariants from a set of counter-examples.

A primary difference in our work, compared to all the work above, is that ours is a *black-box technique* that does not look at the code of the program, but synthesizes an invariant from a snapshot of examples and counter-examples that characterize the invariant. The black-box approach to constructing invariants has both advantages and disadvantages. The main disadvantage is that information regarding what the program actually does is lost in invariant synthesis. However, this is the basis for its advantage as well—by *not* looking at the code, the learning algorithm promises to learn the sets with the simplest representations in polynomial time, and can also be much more flexible. For instance, even when the code of the program is complex, for example having non-linear arithmetic or complex heap manipulations that preclude logical reasoning, black-box learning gives ways to learn simple invariants for them.

There are several black-box learning algorithms that have been explored in verification. Boolean formula learning has been investigated for finding quantifier-free program invariants [17], and also extended to quantified invariants [6]. However, unlike us, [6] learns a quantified formula given a set of data predicates as well as the predicates which can appear in the guards of the quantified formula. Recently, machine learning techniques have also been explored [18]. Variants of the Houdini algorithm [19] essentially use conjunctive Boolean learning (which can be achieved in polynomial time) to learn conjunctive invariants over templates of atomic formulas (see also [20]). The most mature work in this area is Daikon [21], which learns formulas over a template, by enumerating all formulas and checking which ones satisfy the samples, and where scalability is achieved in practice using several heuristics that reduce the enumeration space which is doubly-exponential. For quantified invariants over data-structures, however, such heuristics aren't very effective, and Daikon often restricts learning only to formulas of very restricted syntax, like formulas with a single atomic guard, etc. In our experiments Daikon was, for instance, not able to learn an adequate loop invariant for the selection sort algorithm.

2 Overview

List and Array Invariants: Consider a typical invariant in a sorting program over lists where the loop invariant is expressed as:

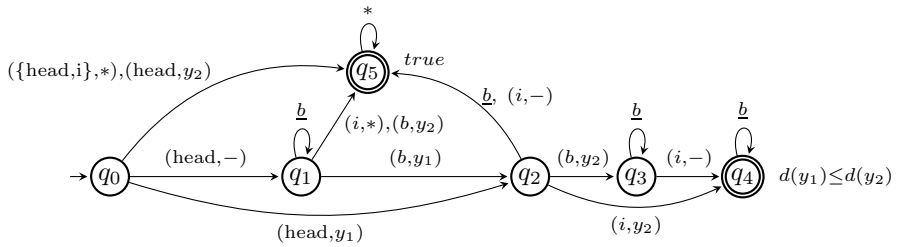
$$head \rightarrow^* i \ \wedge \ \forall y_1, y_2.((head \rightarrow^* y_1 \wedge succ(y_1, y_2) \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \quad (1)$$

This says that for all cells y_1 that occur somewhere in the list pointed to by *head* and where y_2 is the successor of y_1, and where y_1 and y_2 are before the cell pointed to by a scalar pointer variable i, the data value stored at y_1 is no larger than the data value stored at y_2. This formula is *not* in the decidable fragment of STRAND [3,5] since the universally quantified variables are involved in a non-elastic relation *succ* (in the subformula $succ(y_1, y_2)$). Such an invariant for a program manipulating arrays can be expressed as:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_2 = y_1 + 1 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \quad (2)$$

Note that the above formula is not in the decidable array property fragment [4].

Quantified Data Automata: The key idea in this paper is an automaton model for expressing such constraints called *quantified data automata* (QDA). The above two invariants are expressed by the following QDA:



The above automaton reads (deterministically) data words whose labels denote the positions pointed to by the scalar pointer variables *head* and i, as well as valuations of the quantified variables y_1 and y_2. We use two *blank* symbols that indicate that no pointer variable ("b") or no variable from Y ("$-$") is read in the corresponding component; moreover, $\underline{b} = (b, -)$. Missing transitions go to a sink state labeled *false*. The above automaton accepts a data word w with a valuation v for the universally quantified variables y_1 and y_2 as follows: it stores the value of the data at y_1 and y_2 in two registers, and then checks whether the formula annotating the final state it reaches holds for these data values. The automaton accepts the data word w if for *all* possible valuations of y_1 and y_2, the automaton accepts the corresponding word with valuation. The above automaton hence accepts precisely those set of data words that satisfy the invariant formula.

Decidable Fragments and Elastic Quantified Data Automata: The emptiness problem for QDAs is undecidable; in other words, the logical formulas that QDAs express fall into undecidable theories of lists and arrays. A common restriction in the array property fragment as well as the syntactic decidable fragments of STRAND is that quantification is not permitted to be over elements that are only a *bounded* distance away. The restriction allows quantified variables to only be related through *elastic* relations (following the terminology in STRAND [3,5]).

For instance, a formula equivalent to the formula in Eq. 1 but expressed in the decidable fragment of STRAND over lists is:

$$head \rightarrow^* i \ \wedge \ \forall y_1, y_2.((head \rightarrow^* y_1 \wedge y_1 \rightarrow^* y_2 \wedge y_2 \rightarrow^* i) \Rightarrow d(y_1) \leq d(y_2)) \quad (3)$$

This formula compares data at y_1 and y_2 whenever y_2 occurs sometime after y_1, and this makes the formula fall in a decidable class. Similarly, a formula equivalent to the formula Eq. 2 in the decidable array property fragment is:

$$\forall y_1, y_2.((0 \leq y_1 \wedge y_1 \leq y_2 \wedge y_2 \leq i) \Rightarrow A[y_1] \leq A[y_2]) \quad (4)$$

The above two formulas are captured by a QDA that is the same as in the figure above, except that the $\underline{b}$-transition from q_2 to q_5 is replaced by a $\underline{b}$-loop on q_2.

We identify a restricted form of quantified data automata, called *elastic quantified data automata* (EQDA) in Section 5, which structurally captures the constraint that quantified variables can be related only using elastic relations (like $\rightarrow^*$ and $\leq$). Furthermore, we show in Section 6 that EQDAs can be converted to formulas in the decidable fragment of STRAND and the array property fragment, and hence expresses invariants that are amenable to decidable analysis across loop bodies.

It is important to note that QDAs are not necessarily a blown-up version of the formulas they correspond to. For a formula, the corresponding QDA can be exponential, but for a QDA the corresponding formula can be exponential as well (QDAs are like BDDs, where there is sharing of common suffixes of constraints, which is absent in a formula).

3 Quantified Data Automata

We model lists (and finite sets of lists) and arrays that contain data over some data domain D as finite words, called *data words*, encoding the pointer variables and the data values. Consider a finite set of pointer variables $PV = \{p_1, \ldots, p_r\}$ and let $\Sigma = 2^{PV}$. The empty set corresponds to a blank symbol indicating that no pointer variable occurs at this position. We also denote this blank symbol by the letter b. A data word over PV and the data domain D is an element w of $(\Sigma \times D)^*$, such that every $p \in PV$ occurs exactly once in the word (i.e., for each $p \in PV$, there is precisely one j such that $w[j] = (X, d)$, with $p \in X$).

Let us fix a set of variables Y. The automata we build accept a data word if for all possible valuations of Y over the positions of the data word, the data stored at these positions satisfy certain properties. For this purpose, the automaton reads data words extended by valuations of the variables in Y, called valuation words. The variables are then quantified universally in the semantics of the automaton model (as explained later in this section).

A valuation word is a word $v \in (\Sigma \times (Y \cup \{-\}) \times D)^*$, where v projected to the first and third components forms a data word and where each $y \in Y$ occurs in the second component of a letter precisely once in the word. The symbol '$-$' is used for the positions at which no variable from Y occurs. A valuation word hence defines a data word along with a valuation of Y. The data word corresponding to such a word v is the word in $(\Sigma \times D)^*$ obtained by projecting it to its first and third components. Note that the choice of the alphabet enforces the variables from Y to be in different positions.

To express the properties on the data, we fix a set of constants, functions and relations over D. We assume that the quantifier-free first-order theory over this domain is decidable. We encourage the reader to keep in mind the theory of integers with constants (0, 1, etc.), addition, and the usual relations ($\leq$, $<$, etc.) as a standard example of such a domain.

Quantified data automata use a *finite* set F of formulas over the atoms $d(y_1), \ldots, d(y_n)$ that is additionally equipped with a (semi-)lattice structure of the form $\mathcal{F} : (F, \sqsubseteq, \sqcup, \mathit{false}, \mathit{true})$ where $\sqsubseteq$ is the partial-order relation, $\sqcup$ is the least-upper bound, and *false* and *true* are formulas required to be in F and correspond to the bottom and top elements of the lattice. Furthermore, we assume that whenever $\alpha \sqsubseteq \beta$, then $\alpha \Rightarrow \beta$. Also, we assume that each pair of formulas in the lattice are *inequivalent*.

One example of such a formula lattice over the data domain of integers can be obtained by taking a set of representatives of all possible inequivalent Boolean formulas over the atomic formulas involving no constants, defining $\alpha \sqsubseteq \beta$ iff $\alpha \Rightarrow \beta$, and taking the least-upper bound of two formulas as the disjunction of them. Such a lattice would be of size doubly exponential in the number of variables n, and consequently, in practice, we may want to use a different coarser lattice, such as the Cartesian formula lattice. The Cartesian formula lattice is formed over a set of atomic formulas and consists of conjunctions of literals (atoms or negations of atoms). The least-upper bound of two formulas is taken as the conjunction of those literals that occur in both formulas. For the ordering we define $\alpha \sqsubseteq \beta$ if all literals appearing in β also appear in α. The size of a Cartesian lattice is exponential in the number of literals.

We are now ready to introduce the automaton model. A quantified data automaton (QDA) over a set of program variables PV, a data domain D, a set of universally quantified variables Y, and a formula lattice $\mathcal{F}$ is of the form $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\Pi = \Sigma \times (Y \cup \{-\})$, $\delta : Q \times \Pi \to Q$ is the transition function, and $f : Q \to F$ is a *final-evaluation function* that maps each state to a data formula. The alphabet Π used in a QDA does not contain data. Words over Π are referred to as *symbolic words* because they do not contain concrete data values. The symbol $(b, -)$ indicating that a position does not contain any variable is denoted by $\underline{b}$.

Intuitively, a QDA is a *register* automaton that reads the data word extended by a valuation that has a register for each $y \in Y$, which stores the data stored at the positions evaluated for Y, and checks whether the formula decorating the final state reached holds for these registers. It accepts a data word $w \in (\Sigma \times D)^*$ if it accepts *all possible* valuation words v extending w with a valuation over Y.

We formalize this below. A configuration of a QDA is a pair of the form (q, r) where $q \in Q$ and $r : Y \to D$ is a partial variable assignment. The initial configuration is (q_0, r_0) where the domain of r_0 is empty. For any configuration (q, r), any letter $a \in \Sigma$, data value $d \in D$, and variable $y \in Y$ we define $\delta'((q, r), (a, y, d)) = (q', r')$ provided $\delta(q, (a, y)) = q'$ and $r'(y') = r(y')$ for each $y' \neq y$ and $r'(y) = d$, and we let $\delta'((q, r), (a, -, d)) = (q', r)$ if $\delta(q, (a, -)) = q'$. We extend this function δ' to valuation words in the natural way.

A valuation word v is accepted by the QDA if $\delta'((q_0, r_0), v) = (q, r)$ where (q_0, r_0) is the initial configuration and $r \models f(q)$, i.e., the data stored in the registers in the final configuration satisfy the formula annotating the final state reached. We denote the set of valuation words accepted by $\mathcal{A}$ as $L_{\mathrm{v}}(\mathcal{A})$. We assume that a QDA verifies whether its input satisfies the constraints on the number of occurrences of variables from PV and Y, and that all inputs violating these constraints either do not admit a run (because of missing transitions) or are mapped to a state with final formula *false*.

A data word w is accepted by the QDA if every valuation word v that has w as the corresponding data word is accepted by the QDA. The language $L(\mathcal{A})$ of the QDA $\mathcal{A}$ is the set of data words accepted by it.

4 Learning Quantified Data Automata

Our goal in this section is to synthesize QDAs using existing learning algorithms such as Angluin's algorithm [1], which was developed to infer the canonical deterministic automaton for a regular language. We achieve this by relating QDAs to the classical model of Moore machines (an automaton with output on states). Recall that QDAs define two kinds of languages, a language of data words and a language of valuation words. On the level of valuation words, we can view a QDA as a device mapping a symbolic word to a data formula as formalized below.

A *formula word* over PV, $\mathcal{F}$, and Y is an element of $(\Pi^* \times \mathcal{F})$ where, as before, $\Pi = \Sigma \times (Y \cup \{-\})$ and each $p \in PV$ and $y \in Y$ occurs exactly once in the word. Note that a formula word does not contain elements of the data domain—it simply consists of the symbolic word that depicts the pointers into the list (modeled using Σ) and a valuation for the quantified variables in Y (modeled using the second component) as well as a formula over the lattice $\mathcal{F}$. For example, $\big((\{h\}, y_1)(b, -)(b, y_2)(\{t\}, -), d(y_1) \leq d(y_2)\big)$ is a formula word, where h points to the first element, t to the last element, y_1 points to the first element, and y_2 to the third element; and the data formula is $d(y_1) \leq d(y_2)$.

By using formula words we explicitly take the view of a QDA as a Moore machine that reads symbolic words and outputs data formulas. A formula word (u, α) is accepted by a QDA $\mathcal{A}$ if $\mathcal{A}$ reaches the state q after reading u and $f(q) = \alpha$. Hence, a QDA defines a unique language of formula words. One easily observes that two QDAs $\mathcal{A}$ and $\mathcal{A}'$ (over the same lattice of formulas) that accept the same set of valuation words also define the same set of formula words [22] (assuming that all the formulas in the lattice are pairwise non-equivalent).

Thus, a language of valuation words can be seen as a function that assigns to each symbolic word a uniquely determined formula, and a QDA can be viewed as a Moore machine that computes this function. For each such Moore machine there exists a unique minimal one that computes the same function, hence we obtain the following theorem.

Theorem 1. *For each QDA $\mathcal{A}$ there is a unique minimal QDA $\mathcal{A}'$ that accepts the same set of valuation words.*

Angluin [1] introduced a popular learning framework in which a *learner* learns a regular language L, the so-called *target language*, over an a priory fixed alphabet Σ by actively querying a *teacher* which is capable of answering *membership* and *equivalence queries*. Angluin's algorithm learns a regular language in time polynomial in the size of the (unique) minimal deterministic finite automaton accepting the target language and the length of the longest counterexample returned by the teacher.

This algorithm can be easily lifted to the learning of Moore machines. Membership queries now ask for the output or classification of a word. On an equivalence query, the teacher says "yes" or returns a counter-example w such that the output of the conjecture on w is different from the output on w in the target language. Viewing QDAs as Moore machines, we can apply Angluin's algorithm directly in order to learn a QDA, and obtain the following theorem.

Theorem 2. *Given a teacher for a QDA-acceptable language of formula words that can answer membership and equivalence queries, the unique minimal QDA for this language can be learned in time polynomial in this minimal QDA and the length of the longest counterexample returned by the teacher.*

5 Unique Over-approximation Using Elastic QDAs

Our aim is to translate the QDAs that are synthesized into decidable logics such as the decidable fragment of STRAND or the array property fragment. A property shared by both logics is that they cannot test whether two universally quantified variables are bounded distance away. We capture this type of constraint by the subclass of *elastic QDAs (EQDAs)* that have been already informally described in Section 2. Formally, a QDA $\mathcal{A}$ is called *elastic* if each transition on $\underline{b}$ is a self loop, that is, whenever $\delta(q, \underline{b}) = q'$ is defined, then $q = q'$.

The learning algorithm that we use to synthesize QDAs does not construct EQDAs in general. However, we can show that every QDA $\mathcal{A}$ can be *uniquely over-approximated* by a language of valuation words that can be accepted by an EQDA $\mathcal{A}_{\text{el}}$. We will refer to this construction, which we outline below, as *elastification*. This construction crucially relies on the particular structure that elastic automata have, which forces a unique set of words to be added to the language in order to make it elastic.

Let $\mathcal{A} = (Q, q_0, \Pi, \delta, f)$ be a QDA and for a state q let $R_{\underline{b}}(q) := \{q' \mid q \xrightarrow{\underline{b}}^* q'\}$ be the set of states reachable from q by a (possibly empty) sequence of $\underline{b}$-transitions. For a set $S \subseteq Q$ we let $R_{\underline{b}}(S) := \bigcup_{q \in S} R_{\underline{b}}(q)$.

The set of states of $\mathcal{A}_{\text{el}}$ consists of sets of states of $\mathcal{A}$ that are reachable from the initial state $R_{\underline{b}}(q_0)$ of $\mathcal{A}_{\text{el}}$ by the following transition function (where $\delta(S, a)$ denotes the standard extension of the transition function of $\mathcal{A}$ to sets of states):

$$\delta_{\text{el}}(S, a) = \begin{cases} R_{\underline{b}}(\delta(S, a)) & \text{if } a \neq \underline{b} \\ S & \text{if } a = \underline{b} \text{ and } \delta(q, \underline{b}) \text{ is defined for some } q \in S \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Note that this construction is similar to the usual powerset construction except

that in each step we take the $\underline{b}$-closure after applying the transition function of $\mathcal{A}$. If the input letter is $\underline{b}$, $\mathcal{A}_{\text{el}}$ loops on the current set if a $\underline{b}$-transition is defined for some state in the set.

The final evaluation formula for a set is the least upper bound of the formulas for the states in the set: $f_{\text{el}}(S) = \bigsqcup_{q \in S} f(q)$. We can now show that $L_{\text{v}}(\mathcal{A}_{\text{el}})$ is the *most precise elastic over-approximation* of $L_{\text{v}}(\mathcal{A})$.

Theorem 3. *For every QDA $\mathcal{A}$, the EQDA $\mathcal{A}_{el}$ satisfies $L_v(\mathcal{A}) \subseteq L_v(\mathcal{A}_{el})$, and for every EQDA $\mathcal{B}$ such that $L_v(\mathcal{A}) \subseteq L_v(\mathcal{B})$, $L_v(\mathcal{A}_{el}) \subseteq L_v(\mathcal{B})$ holds.*

Proof: Note that $\mathcal{A}_{\text{el}}$ is elastic by definition of δ_{el}. It is also clear that $L_{\text{v}}(\mathcal{A}) \subseteq L_{\text{v}}(\mathcal{A}_{\text{el}})$ because for each run of $\mathcal{A}$ using states $q_0 \cdots q_n$ the run of $\mathcal{A}_{\text{el}}$ on the same input uses sets $S_0 \cdots S_n$ such that $q_i \in S_i$, and by definition $f(q_n)$ implies $f_{\text{el}}(S_n)$.

Now let $\mathcal{B}$ be an EQDA with $L_{\text{v}}(\mathcal{A}) \subseteq L_{\text{v}}(\mathcal{B})$. Let $w = (a_1, d_1) \cdots (a_n, d_n) \in L_{\text{v}}(\mathcal{A}_{\text{el}})$ and let S be the state of $\mathcal{A}_{\text{el}}$ reached on w. We want to show that $w \in L_{\text{v}}(\mathcal{B})$. Let p be the state reached in $\mathcal{B}$ on w. We show that $f(q)$ implies $f_{\mathcal{B}}(p)$ for each $q \in S$. From this we obtain $f_{\text{el}}(S) \Rightarrow f_{\mathcal{B}}(p)$ because $f_{\text{el}}(S)$ is the least formula that is implied by all the $f(q)$, for $q \in S$.

Pick some state $q \in S$. By definition of δ_{el} we can construct a valuation word $w' \in L_{\text{v}}(\mathcal{A})$ that leads to the state q in $\mathcal{A}$ and has the following property: if all letters of the form $(\underline{b}, d)$ are removed from w and from w', then the two remaining words have the same symbolic words. In other words, w and w' can be obtained from each other by inserting and/or removing $\underline{b}$-letters.

Since $\mathcal{B}$ is elastic, w' also leads to p in $\mathcal{B}$. From this we can conclude that $f(q) \Rightarrow f_{\mathcal{B}}(p)$ because otherwise there would be a model of $f(q)$ that is not a model of $f_{\mathcal{B}}(p)$ and by changing the data values in w' accordingly we could produce an input that is accepted by $\mathcal{A}$ and not by $\mathcal{B}$. □

6 Linear Data-Structures to Words and EQDAs to Logics

In this section, we sketch briefly how to model arrays and lists as data words, and how to convert EQDAs to quantified logical formulas in decidable logics.

Modeling Lists and Arrays as Data Words: We model a linear data structure as a word over $(\Sigma \times D)$ where $\Sigma = 2^{PV}$, PV is the set of pointer variables and D is the data domain; scalar variables in the program are modeled as single element lists. The encoding introduces a special pointer variable *nil* which is always read together with all other null-pointers in the configuration. For arrays, the encoding introduces variables *le_zero* and *geq_size* which are read together with all those index variables which are less than zero or which exceed the size of the respective array. Given a configuration, the corresponding data words read the scalar variables and the linear data structures one after the other, in some pre-determined order. In programs like copying one array to another, where both the arrays are read synchronously, the encoding models multiple data structures as a single structure over an extended data domain.

From EQDAs to STRAND and Array Property Fragment (APF): Now we briefly sketch the translation from an EQDA $\mathcal{A}$ to an equivalent formula $\mathcal{T}(\mathcal{A})$ in STRAND or the APF such that the set of data words accepted by $\mathcal{A}$ corresponds to the program configurations $\mathcal{C}$ which model $\mathcal{T}(\mathcal{A})$.

Given an EQDA $\mathcal{A}$, the translation enumerates all simple paths in the automaton to an output state. For each such path p from the initial state to an output state q_p, the translation records the relative positions of the pointer and universal variables as a structural constraint ϕ_p, and the formula $f_{\mathcal{A}}(q_p)$ relating the data value at these positions. Each path thus leads to a universally quantified implication of the form $\forall Y.\ \phi_p \Rightarrow f_{\mathcal{A}}(q_p)$. All valuation words not accepted by the EQDA semantically go to the formula *false*, hence an additional conjunct $\forall Y.\ \neg(\bigvee_p \phi_p) \Rightarrow \mathit{false}$ is added to the formula. So the final formula is $\mathcal{T}(\mathcal{A}) = \big(\bigwedge_p \forall Y.\ \phi_p \Rightarrow f_{\mathcal{A}}(q_p)\big) \ \wedge\ \big(\forall Y.\ \neg(\bigvee_p \phi_p) \Rightarrow \mathit{false}\big)$.

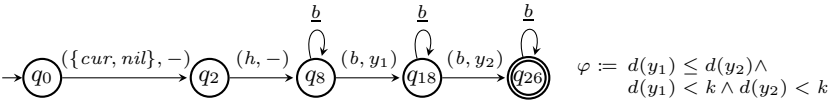


Fig. 1. A path in the automaton expressing the invariant of the program which finds a key k in a sorted list.The full automaton is presented in [22].

We next explain, through an example, the construction of the structural constraints ϕ_p (for details see [22]). Consider program *list-sorted-find* which searches for a key in a sorted list. The EQDA corresponding to the loop invariant learned for this program is presented in [22]. One of the simple paths in the automaton (along with the associated self-loops on $\underline{b}$) is shown in Fig 1. The structural constraint ϕ_p intuitively captures all valuation words which are accepted by the automaton along p; for the path in the figure ϕ_p is $(\mathit{cur} = \mathit{nil} \wedge h \rightarrow^+ y_1 \wedge y_1 \rightarrow^+ y_2)$ and the formula $\forall y_1 y_2.\ (\mathit{cur} = \mathit{nil} \wedge h \rightarrow^+ y_1 \wedge y_1 \rightarrow^+ y_2) \Rightarrow (d(y_1) \leq d(y_2) \wedge d(y_1) < k \wedge d(y_2) < k)$ is the corresponding conjunct in the learned invariant. Applying this construction yields the following theorem.

Theorem 4. *Let $\mathcal{A}$ be an EQDA, w a data word, and c the program configuration corresponding to w. If $w \in \mathcal{L}(\mathcal{A})$, then $c \models \mathcal{T}(\mathcal{A})$. Additionally, if $\mathcal{T}(\mathcal{A})$ is a* STRAND *formula, then the implication also holds in the opposite direction.*

APF allows the universal variables to be related by $\leq$ or $=$ and not $<$. Hence, along paths where $y_1 < y_2$, we over-approximate the structural constraint ϕ_p to $y_1 \leq y_2$ and, subsequently, the data formula $f_{\mathcal{A}}(q_p)$ is abstracted to include $d(y_1) = d(y_2)$. This leads to an abstraction of the actual semantics of the EQDA and is the reason Theorem 4 only holds in one direction for the APF.

7 Implementation and Evaluation on Learning Invariants

We apply the active learning algorithm for QDAs, described in Section 4, in a passive learning framework in order to learn quantified invariants over lists and arrays from a finite set of samples S obtained from dynamic test runs.

Implementing the Teacher: In an active learning algorithm, the learner can query the teacher for membership and equivalence queries. In order to build a passive learning algorithm from a sample S, we build a teacher, who will use S to answer the questions of the learner, ensuring that the learned set contains S.

The teacher knows S and wants the learner to construct a small automaton that includes S; however, the teacher does not have a particular language of data words in mind, and hence cannot answer questions precisely. We build a teacher who answers queries as follows: On a membership query for a word w, the teacher checks whether w belongs to S and returns the corresponding data formula. The teacher has no knowledge about the membership for words which were not realized in test runs, and she rejects these. She also does not know whether the formula she computes on words that get manifest can be weaker; but she insists on that formula. By doing these, the teacher errs on the side of keeping the invariant semantically small. On an equivalence query, the teacher just checks that the set of samples S is contained in the conjectured invariant. If not, the teacher returns a counter-example from S. Note that the passive learning algorithm hence guarantees that the automaton learned will be a superset of S and will take polynomial time in the learnt automaton. We show the efficacy of this passive learning algorithm using experimental evidence.

Implementation of a Passive Learner of Invariants: We first take a program and using a test suite, extract the set of concrete data-structures that get manifest at loop-headers (for learning loop invariants) and at the beginning/end of functions (for learning pre/post conditions). The test suite was generated by enumerating all possible arrays/lists of a small bounded length, and with data-values from a small bounded domain. We then convert the data-structures into a set of formula words, as described below, to get the set S on which we perform passive learning. We first fix the formula lattice $\mathcal{F}$ over data formulas to be the Cartesian lattice of atomic formulas over relations $\{=, <, \leq\}$. This is sufficient to capture the invariants of many interesting programs such as sorting routines, searching a list, in-place reversal of sorted lists, etc. Using lattice $\mathcal{F}$, for every program configuration which was realized in some test run, we generate a formula word for every valuation of the universal variables over the program structures. We represent these formula words as a mapping from the symbolic word, encoding the structure, to a data formula in the lattice $\mathcal{F}$. If different inputs realize the same structure but with different data formulas, we associate the symbolic word with the join of the two formulas.

Implementing the Learner: We used the libALF library [23] as an implementation of the active learning algorithm [1]. We adapted its implementation to our setting by modeling QDAs as Moore machines. If the learned QDA is not elastic, we elastify it as described in Section 5. The result is then converted to a quantified formula over Strand or the APF and we check if the learned invariant was adequate using a constraint solver.

Table 1. Results of our experiments

Example	LOC	#Test inputs	$T_{teacher}$ (s)	#Eq.	#Mem.	Size #states	Elastification required ?	T_{learn} (s)
array-find	25	310	0.05	2	121	8	no	0.00
array-copy	25	7380	1.75	2	146	10	no	0.00
array-compare	25	7380	0.51	2	146	10	no	0.00
insertion-sort-outer	30	363	0.19	3	305	11	no	0.00
insertion-sort-innner	30	363	0.30	7	2893	23	yes	0.01
selection-sort-outer	40	363	0.18	3	306	11	no	0.01
selection-sort-inner	40	363	0.55	9	6638	40	yes	0.05
list-sorted-find	20	111	0.04	6	1683	15	yes	0.01
list-sorted-insert	30	111	0.04	3	1096	20	no	0.01
list-init	20	310	0.07	5	879	10	yes	0.01
list-max	25	363	0.08	7	1608	14	yes	0.00
list-sorted-merge	60	5004	10.50	7	5775	42	no	0.06
list-partition	70	16395	11.40	10	11807	38	yes	0.11
list-sorted-reverse	25	27	0.02	2	439	18	no	0.00
list-bubble-sort	40	363	0.19	3	447	12	no	0.01
list-fold-split	35	1815	0.21	2	287	14	no	0.00
list-quick-sort	100	363	0.03	1	37	5	no	0.00
list-init-complex	80	363	0.05	1	57	6	no	0.01
lookup_prev	25	111	0.04	3	1096	20	no	0.01
add_cachepage	40	716	0.19	2	500	14	no	0.01
Glib sort (merge)	55	363	0.04	1	37	5	no	0.00
Glib insert_sorted	50	111	0.04	2	530	15	no	0.01
devres	25	372	0.06	2	121	8	no	0.00
rm_pkey	30	372	0.06	2	121	8	no	0.00
GNU Coreutils sort	2500	1 File	0.00	17	4996	5	yes	0.07
Learning Function Pre-conditions								
list-sorted-find	20	111	0.01	1	37	5	no	0.00
list-init	20	310	0.02	1	26	4	no	0.00
list-sorted-merge	60	329	0.06	3	683	19	no	0.01

Experimental Results:[1] We evaluate our approach on a suite of programs (see Table 1) for learning invariants and preconditions. For every program, we report the number of lines of C code, the number of test inputs and the time ($T_{teacher}$) taken to build the teacher from the samples collected along these test runs. We also report the number of equivalence and membership queries answered by the teacher in the active learning algorithm, the size of the final elastic automata, whether the learned QDA required any elastification and finally, the time (T_{learn}) taken to learn the QDA.

[1] More details at `http://web.engr.illinois.edu/~garg11/learning_qda.html`

The names of the programs in Table 1 are self-descriptive and we only describe some of them. The *inner* and *outer* suffix in insertion and selection sort corresponds to learning loop-invariants for the inner and outer loops in those sorting algorithms. The program *list-init-complex* sorts an input array using heap-sort and then initializes a list with the contents of this sorted array. Since heap-sort is a complex algorithm that views an array as a binary tree, none of the current automatic white-box techniques for invariant synthesis can handle such complex programs. However, our learning approach being black-box, we are able to learn the correct invariant, which is that the list is sorted. Similarly, synthesizing post-condition annotations for recursive procedures like merge-sort and quick-sort is in general difficult for white-box techniques, like *interpolation*, which require a post-condition. In fact, SAFARI [14], which is based on *interpolation*, cannot handle list-structures, and also cannot handle array-based programs with *quantified* preconditions which precludes verifying the array variants of programs like *sorted-find*, *sorted-insert*, etc., which we can handle.

The methods *lookup_prev* and *add_cachepage* are from the module cachePage in a verified-for-security platform for mobile applications [24]. The module cachePage maintains a cache of the recently used disc pages as a priority queue based on a sorted list. The method *sort* is a merge sort implementation and *insert_sorted* is a method for insertion into a sorted list. Both these methods are from Glib which is a low-level C library that forms the basis of the GTK+ toolkit and the GNOME environment. The methods *devres* and *rm_pkey* are methods adapted from the Linux kernel and an Infiniband device driver, both mentioned in [6]. Finally, we learn the sortedness property (with respect to the method *compare* that compares two lines) of the method *sortlines* which lies at the heart of the GNU core utility to sort a file. The time taken by our technique to learn an invariant, being black-box, largely depends on the complexity of the property and not the size of the code, as is evident from the successful application of our technique to this large program.

All experiments were completed on an Intel Core i5 CPU at 2.4GHz with 6GB of RAM. For all examples, our prototype implementation learns an adequate invariant really fast. Though the learned QDA might not be the smallest automaton representing the samples S (because of the inaccuracies of the teacher), in practice we find that they are reasonably small (fewer than 50 states). Moreover, we verified that the learned invariants were adequate for proving the programs correct by generating verification conditions and validating them using an SMT solver (these verified in less than 1s). It is possible that SMT solvers can sometimes even handle non-elastic invariants and VCs; however, in our experiments, it was not able to handle such formulas without giving extra triggers, thus suggesting the necessity of the elastification of QDAs. Learnt invariants are complex in some programs; for example the invariant QDA for the program *list-sorted-find* is presented in [22] and corresponds to:

$head \neq nil \wedge (\forall y_1 y_2.head \rightarrow^* y_1 \rightarrow^* y_2 \Rightarrow d(y_1) \leq d(y_2)) \wedge ((cur = nil \wedge \forall y_1.head \rightarrow^* y_1 \Rightarrow d(y_1) < k) \vee (head \rightarrow^* cur \wedge \forall y_1.head \rightarrow^* y_1 \rightarrow^+ cur \Rightarrow d(y_1) < k))$.

Future Work: We believe that learning of structural conditions of data-structure invariants using automata is an effective technique, especially for quantified properties where passive or machine-learning techniques are not currently known. However, for the data-formulas themselves, machine learning can be very effective [18], and we would like to explore combining automata-based structural learning (for words and trees) with machine-learning for data-formulas.

Acknowledgements. We would like to thank Xiaokang Qiu for valuable discussions on the automata model for STRAND formulas. This work is partially supported by NSF CAREER award #0747041.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
2. Kearns, M.J., Vazirani, U.V.: An introduction to computational learning theory. MIT Press, Cambridge (1994)
3. Madhusudan, P., Parlato, G., Qiu, X.: Decidable logics combining heap structures and data. In: POPL, pp. 611–622. ACM (2011)
4. Bradley, A.R., Manna, Z., Sipma, H.B.: What's decidable about arrays? In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 427–442. Springer, Heidelberg (2006)
5. Madhusudan, P., Qiu, X.: Efficient decision procedures for heaps using STRAND. In: Yahav, E. (ed.) SAS 2011. LNCS, vol. 6887, pp. 43–59. Springer, Heidelberg (2011)
6. Kong, S., Jung, Y., David, C., Wang, B.-Y., Yi, K.: Automatically inferring quantified loop invariants by algorithmic learning from simple templates. In: Ueda, K. (ed.) APLAS 2010. LNCS, vol. 6461, pp. 328–343. Springer, Heidelberg (2010)
7. Cobleigh, J.M., Giannakopoulou, D., Păsăreanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
8. Alur, R., Madhusudan, P., Nam, W.: Symbolic compositional verification by learning assumptions. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 548–562. Springer, Heidelberg (2005)
9. Sagiv, M., Reps, T., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
10. Bouajjani, A., Drăgoi, C., Enea, C., Sighireanu, M.: Abstract domains for automated reasoning about list-manipulating programs with infinite data. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 1–22. Springer, Heidelberg (2012)
11. McMillan, K.L.: Interpolation and SAT-Based model checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
12. Jhala, R., McMillan, K.L.: Array abstractions from proofs. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 193–206. Springer, Heidelberg (2007)
13. McMillan, K.L.: Quantified invariant generation using an interpolating saturation prover. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 413–427. Springer, Heidelberg (2008)

14. Alberti, F., Bruttomesso, R., Ghilardi, S., Ranise, S., Sharygina, N.: Safari: Smt-based abstraction for arrays with interpolants. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 679–685. Springer, Heidelberg (2012)
15. Seghir, M.N., Podelski, A., Wies, T.: Abstraction refinement for quantified array assertions. In: Palsberg, J., Su, Z. (eds.) SAS 2009. LNCS, vol. 5673, pp. 3–18. Springer, Heidelberg (2009)
16. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
17. Chen, Y.F., Wang, B.Y.: Learning boolean functions incrementally. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 55–70. Springer, Heidelberg (2012)
18. Sharma, R., Nori, A.V., Aiken, A.: Interpolants as classifiers. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 71–87. Springer, Heidelberg (2012)
19. Flanagan, C., Leino, K.R.M.: Houdini, an annotation assistant for eSC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
20. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI, pp. 223–234. ACM (2009)
21. Ernst, M.D., Czeisler, A., Griswold, W.G., Notkin, D.: Quickly detecting relevant program invariants. In: ICSE, pp. 449–458 (2000)
22. Garg, P., Löding, C., Madhusudan, P., Neider, D.: Learning universally quantified invariants of linear data structures. CoRR abs/1302.2273 (2013)
23. Bollig, B., Katoen, J.-P., Kern, C., Leucker, M., Neider, D., Piegdon, D.R.: libalf: The Automata Learning Framework. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 360–364. Springer, Heidelberg (2010)
24. Mai, H., Pek, E., Xue, H., King, S.T., Madhusudan, P.: Verifying security invariants in ExpressOS. In: ASPLOS, pp. 293–304 (2013)

Towards Distributed Software Model-Checking Using Decision Diagrams*

Maximilien Colange[1], Souheib Baarir[2], Fabrice Kordon[1], and Yann Thierry-Mieg[1]

[1] LIP6, CNRS UMR 7606, Université P. & M. Curie – Paris 6
4, Place Jussieu, F-75252 Paris Cedex 05, France
[2] LIP6, CNRS UMR 7606 and Université Paris Ouest Nanterre La Défense
200, avenue de la République, F-92001 Nanterre Cedex, France
first.last@lip6.fr

Abstract. Symbolic data structures such as Decision Diagrams have proved successful for model-checking. For high-level specifications such as those used in programming languages, especially when manipulating pointers or arrays, building and evaluating the transition is a challenging problem that limits wider applicability of symbolic methods.

We propose a new symbolic algorithm, *EquivSplit*, allowing an efficient and fully symbolic manipulation of transition relations on Data Decision Diagrams. It allows to work with equivalence classes of states rather than individual states. Experimental evidence on the concurrent software oriented benchmark BEEM shows that this approach is competitive.

1 Introduction

Model-checking of concurrent software faces state space explosion. To address this issue, many algorithms and data structures have been proposed, one of the most successful being symbolic shared data structures such as Binary Decision Diagrams (BDD).

While BDD allow in many cases to cope with very large state spaces, expressing algorithms symbolically to take full advantage of the data structure is tricky. Symbolic evaluation algorithms that are aware of the data structure itself such as saturation-style algorithms [6,11] can be orders of magnitude better than naive evaluation in a breadth-first search manner.

The transition relation of a system of k boolean variables, can be seen as a function $\mathbb{B}^k \mapsto 2^{\mathbb{B}^k}$ and is usually built and stored as a second decision diagram N, with two variables "before" and "after" for each variable of the system. A specific operation between any subset of the state space S encoded as a decision diagram and the transition relation N yields a decision diagram $S' = N(S)$ representing immediate successors of S.

Let us define statements as (sequences of) assignments of expressions to variables. The support of a statement is the set of variables it reads or writes to. This notion of locality is heavily exploited, to limit the representation of transitions to the effect they have on variables of their support. For each transition with k' Boolean support variables,

* This work has been supported by a grant from the Délégation Générale pour l'Armement and by the project ImpRo/ANR-2010-BLAN-0317.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 830–845, 2013.

worst case representation size is $2^{k'}$. The symbolic approach was successfully applied to Boolean gate logic where encoding these $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices is feasible.

But because classical approaches compute potential to potential $\mathbb{B}^{k'} \mapsto 2^{\mathbb{B}^{k'}}$ transition matrices, a larger support for transitions means exponential growth of the worst case complexity in representation size. It also severely limits the possibilities of saturation-based techniques as their efficiency relies in clusters based on the support of transitions. Hence, a worst case for classical symbolic approaches is when the support of transitions includes all variables.

Moreover, when the input specification includes array or pointer manipulation, any static analysis of statements will necessarily yield pessimistic support assumptions. For instance, a non-constant array access such as `t[i]` *may* depend on the variable `t[0]`. In classical approaches, pessimistic assumptions must include all elements of the array t in the support. Such expressions are commonly encountered in modeling languages such as Promela or Divine [12,2].

We propose in this paper to perform a dynamic analysis of such statements as they are being resolved, allowing to discover more locality in the remaining effects as expressions are partially evaluated. This can avoid the problems induced by transitions with a large syntactic support by only performing the computations that are *really necessary*. Our algorithm exploits locality to optimize its evaluation, as the support of expressions may vary as the evaluation progresses.

In the dynamic case, when evaluating `t[i]`, as soon as the value of the index expression `i` has been reduced to a constant, pessimistic assumptions can be forgotten and the support is reduced to the effective cell of the array that is the target of the assignment.

To have efficient symbolic computations of these statements, we define an equivalence relation over states with respect to the value of an expression; this induces equivalence classes that can be built dynamically and manipulated symbolically. Intuitively, if efficient manipulation of equivalence classes is possible, then the computation complexity can be proportional to the number of such equivalence classes rather than to the number of actual states.

We define in this paper a new decision diagram based operation, *EquivSplit*, that allows to efficiently compute and manipulate such equivalence classes, in a way compatible with the decision diagram encoding of states. Given a syntax tree e for an arbitrary expression, and a set of states S encoded as a decision diagram, we provide an incremental and on the fly algorithm to efficiently compute a partition of $S = S_0 \uplus S_1 \uplus \ldots$ where all states in a S_i agree on the value of e, and no two distinct S_i, S_j agree on the value of e.

Outline. We first introduce notations for expressions and their (partial) evaluation. We then recall the definition of Data Decision Diagrams (DDD)[10], as the type of integer valued decision diagrams we use in our implementation. We then explain the *EquivSplit* algorithm and how it is used to evaluate and resolve expressions on sets of values stored as DDD. To assess the applicability of our approach in practice, we study in section 5 the efficiency of our approach for Divine models taken from a standard benchmark (BEEM) and compare it to other symbolic approaches.

2 Expressions

We first define in 2.1 some concepts and introduce notations that will be used throughout the paper. The abstract level of these definitions guarantees independence from any concrete syntax. We give flesh to these definitions with more concrete examples in 2.2.

2.1 Definitions and Notations

Let Σ be a signature, that is a set of symbols of finite arity. We inductively define the set `Expr` of Σ-*expressions* as $\phi \in$ `Expr` if and only if:

- $\phi \in \Sigma$ of arity 0,
- or $\phi = s(\phi_1, \ldots, \phi_k)$ where $s \in \Sigma$ is of arity k and $\phi_1, \ldots, \phi_k \in Expr$ (ϕ_i is called a sub-expression).

Let D be a domain for expressions. We assume that D is embedded in Σ, so that every element of the domain can be referred to syntactically.

Definition 1. *An* interpretation *I is a function that associates to every symbol $s \in \Sigma$ of arity $k > 0$ a (possibly partial) function $I(s) : D^k \mapsto D$, and that maps each symbol of arity 0 to its corresponding element of D.*

Intuitively, this formalism captures most programming languages, with pointers and pointer arithmetic. From now on, we assume that there is a finite subset X in D, called *addresses*. The set of addresses X being finite, we note $X = \{x_1, \ldots, x_{|X|}\}$. We assume Σ contains a special symbol δ of arity 1, that allows to access a memory slot given its address. Note that a variable is just a symbolic name for an address. Thus, $I(\delta)$ represents the content of the memory that varies as the program runs. Since we focus on the evolution of the content of the memory, all the interpretations considered from now on are equal for the other symbols (i.e. the operational semantics for the symbols of the language is known and fixed). Let $\mu = I(\delta)$ designate a *valuation*, i.e. the state of the memory. μ is seen as a (partial, when not all memory contents are known) function from X into D. Since all other symbols have a fixed interpretation, an interpretation I can be described by simply providing μ. Furthermore, all symbols interpretations must be complete functions (only the valuation is allowed to be a partial function). Partial interpretations can be completed by adding a special element to D and mapping the undefined domain onto this special element. This special element corresponds to an error or an undefined behavior. Note that the interpretations of all symbols must take into account this new special element.

Definition 2. *Given an interpretation I, an expression $\phi = s(\phi_1, \ldots, \phi_k)$ ($k \geq 0$)* evaluates *or* reduces *to another expression $eval(I, \phi)$ as follows:*

$$eval(I,\phi) = \begin{cases} I(s) \in D & \text{if } s \text{ is a symbol of arity } 0 \\ I(s)(eval(I,\phi_1), \ldots, eval(I,\phi_k)) \in D & \text{if } eval(I,\phi_i) \in D \text{ for all } i \text{ and } I(s) \text{ is defined at this point} \\ s(eval(I,\phi_1), \ldots, eval(I,\phi_k)) & \text{otherwise.} \end{cases}$$

If $eval(I, \phi) \in D$, the evaluation is complete.

Notation. We will now abusively denote the evaluation $eval(I,\phi)$ where $I(\delta)=\mu$ by $eval(\mu,\phi)$. If ψ is a (possibly nested) sub-expression of ϕ, $\phi[\psi \leftarrow \theta]$ denotes the expression obtained by substituting the expression θ to ψ in ϕ. Given a valuation μ and a subset of addresses $Y \subseteq X$, $\mu_{|Y}$ denotes the restriction of μ to Y. With these notations, we have, for any variable x, any valuation μ where x is defined, and any expression ϕ: $\phi[\delta(x) \leftarrow \mu(x)] = eval(\mu_{|\{x\}},\phi)$

We now define an equivalence relation on valuations with respect to the evaluation of an expression. In Section 4 this equivalence relation is a key notion, allowing efficient evaluation of expressions on sets of valuations.

Definition 3. *Given a subset Y of X and an expression ϕ, for all valuations μ,μ' we define the equivalence relation $\sim_\phi^Y$ as follows:*

$$\mu \sim_\phi^Y \mu' \Leftrightarrow eval(\mu_{|Y},\phi) = eval(\mu'_{|Y},\phi)$$

A trivial case of this equivalence is valuations $\mu \neq \mu'$, that are equal on Y.

2.2 Examples of Expressions

To help in visualizing these definitions, let us use as an example a language supporting a C-like syntax. We give concrete examples here for each element defined abstractly above. We consider a language supporting integers and their manipulation operators (arithmetic +, -, * ... as well as bitwise operations <<,>>,...). The set of considered operators are part of the signature Σ. The domain D is thus integers. The Σ-expressions are built by syntactic combinations of operators, and the literals 0 or 1 are also (terminal) expressions (as D is embedded in Σ).

Then, by definition 1, we must provide an interpretation function I that gives the semantics of all the operators which are used in expressions. The interpretation function works with constants; for our example we should provide the integer output value for each of the binary operators given two integers.

Consider now variables of the program "a,b,c". They are seen as symbolic names and mapped to integers (memory addresses), for instance $0,1,2$. The special operator δ allows to read the value of such a variable, hence the expression a is interpreted as $\delta(0)$. We add the notion of array of fixed size *tab*, and access to a cell of an array using $tab[]$. Again *tab* is a symbolic name for a variable mapped to an integer, for instance 3 that is the first memory slot occupied by the array. Then $tab[e]$ where e is an arbitrary expression is a syntactic sugar for $\delta(3+e)$.

All operators should have complete interpretations: a/b must also be defined when $b=0$. For this purpose, one or more special constants can be introduced. For a given language manipulating finite types, the definition of the interpretation of most symbols is usually straightforward. We consider that the interpretation of all symbols except δ is fixed throughout the computations. In other words we distinguish the code (all other symbols from the signature) from the data, represented by $I(\delta)$, that may vary as the computation progresses.

Definition 2 formalizes partial evaluation of expressions given an interpretation function. For instance, suppose μ only gives the content of memory slot 0, say $\mu(0)=12$.

Let $\phi = add(\delta(0), \delta(1))$ (usually noted $a+b$). Then $eval(\mu, \phi) = add(eval(\mu, \delta(0)), eval(\mu, \delta(1)))$. We have $eval(\mu, \delta(0)) = I(\delta)(0) = \mu(0) = 12$. However, because μ is not defined for address 1, $eval(\mu, \delta(1)) = \delta(1)$. Hence, $eval(\mu, \phi) = add(12, \delta(1))$ (noted $12+b$).

As an example for Definition 3, any two μ, μ' such that $eval(\mu, a+b) = eval(\mu', a+b)$ are equivalent. For instance, if both a and b are in Y, $\mu = (a \leftarrow 0, b \leftarrow 1), \mu' = (a \leftarrow 1, b \leftarrow 0)$ are equivalent. If only a is in Y, μ and μ' are not equivalent, since one yields expression $0+b$ while the other yields $1+b$.

3 Data Decision Diagrams (DDD) [10]

Let us now briefly recall important concepts of decision diagrams. The algorithm presented in this paper is valid for any type of shared decision diagram, such as BDD. However, to more closely match our definition of expressions, we will consider here Data Decision Diagrams, where the domain of variables is D rather than $\mathbb{B}$. This provides a natural representation for a set of valuations as a DDD.

Shared Decision Diagrams (DD) are a data structure to compactly represent sets. There are many variants of decision diagrams used for model-checking, but they all rely on the same underlying principles: nodes of the decision tree are unique in memory thanks to a canonical representation; the number of paths through the diagram (states) can be exponential in the representation size (nodes in the DD); equality of two sets can be tested in constant time; using caches most operations manipulating a DD are polynomial in the representation size; the effectiveness of the encoding strongly depends on the chosen variable ordering [7].

In this paper we rely on Data Decision Diagrams (DDD, defined in [10]), which extend classical BDD in two respects: *1)* variables are considered to have an integer domain instead of a Boolean one, and, *2)* operations over DDD are encoded using homomorphisms instead of the usual fashion where another decision diagram with two variables per variable of the state signature is used.

A DDD is a data structure for representing a set of sequences of assignments of the form $x_1 := v_1; x_2 := v_2; \ldots; x_n := v_n$, where x_i are variables and v_i are values in D. We assume a total order on variables such that all variables are always encountered in the same order in an assignment sequence. The usual DDD definition makes weaker assumptions on variable ordering, but these are out of the scope of this paper (see [10]).

We define the terminal **1** to represent the empty assignment sequence, that terminates any valid sequence, and **0** to represent the empty set of assignment sequences.

Definition 4 (DDD). *Let X be a set of variables ranging over domain D. The set $\mathbb{D}$ of DDD is defined inductively by:*
$\delta \in \mathbb{D}$ *if either* $\delta \in \{\mathbf{0}, \mathbf{1}\}$ *or* $\delta = \langle x, \alpha \rangle$ *with* $x \in X$, *and* $\alpha : D \to \mathbb{D}$ *is a mapping where only a finite subset of D maps to other DDD than* **0**.

By convention, edges that map to the DDD **0** *are not represented.*

For instance, consider the DDD shown in figure 1. Each path in the DDD corresponds to a sequence of assignments. In this work, we use DDD to represent valuations of the memory, thus each assignment sequence represents a memory state.

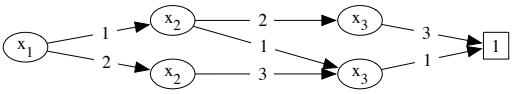


Fig. 1. This DDD with domain $D = \mathbb{N}$ represents the set of sequences of assignments: $\{(x_1 := 2; x_2 := 3; x_3 := 1;), (x_1 := 1; x_2 := 1; x_3 := 1;), (x_1 := 1; x_2 := 2; x_3 := 3;)\}$

Operations and Homomorphisms. DDD support standard set operations: $\cup$, $\cap$, $\setminus$. The semantics of these operations are based on the sets of assignment sequences that the DDD represent.

Basic and inductive homomorphisms are also introduced to define application specific operations. A detailed description of DDD homomorphisms can be found in [10].

Since in this paper we define new symbolic operations that are not specific to DDD, we omit further details on homomorphisms. From the implementation point of view, all operations we define are embedded in homomorphisms. This allows the software library to enable automatic rewritings that yield much better performances, such as the saturation algorithm [11].

4 Evaluating Expressions on DDD

In practice, a system's state is a valuation of the state variables, and the behavior of the system is described with expressions. When treating such a system using DDD arises the need to evaluate an expression over a *set* of valuations.

More precisely, given an expression ϕ and a set of valuations V, one needs to compute all the evaluations of ϕ by the valuations in V. To achieve this goal efficiently, we rely on equivalence relation $\sim_\phi^X$ of definition 3.

Recall that the size of a DDD is often logarithmic in the size of the represented set. The naive approach considers each valuation separately, ending up with a complexity linear in the size of the input set. An efficient solution to this problem should use functions that manipulate the nodes of the data structure representation, so that thanks to caches, the complexity remains proportional to the encoding size.

We propose an algorithm, *EquivSplit*, that partitions a set of valuations (given as a DDD) into equivalence classes with respect to $\sim_\phi^X$. It visits variables in the order given by the DDD, and progressively evaluates the expression. Hence it must work with partial valuations and partially evaluated expressions.

We first define in section 4.1 the notion of dependency on an address, and how to resolve such dependencies to ensure proper recursion. We then present our algorithm in a restricted case to help comprehension in section 4.2. It is extended to the general case, by introducing another function *SolveSub* in section 4.3. The correction and the complexity of these functions are discussed in section 4.4.

4.1 Support of Expressions

The support of an expression is the set of memory addresses necessary to completely evaluate this expression. Conversely, an expression does not depend on an address if

its content does not affect its evaluation. We formally define these notions, and then explain how to partially evaluate an expression until dependencies on a given address are eliminated.

Definition 5. *An expression* ϕ does not depend *on an address* x *if and only if:*

$$\forall \mu, \mu' \in D^X, \mu_{|X\setminus\{x\}} = \mu'_{|X\setminus\{x\}} \implies eval(\mu, \phi) = eval(\mu', \phi)$$

The support *of an expression* ϕ *is the set of addresses on which* ϕ *depends.*
An expression that depends on no variable is said to be constant.

Lemma 1. *If* ϕ *is an expression that depends on* x*, then there exist a sub-expression* $\delta(\psi)$ *of* ϕ *and a valuation* μ *such that* $eval(\mu, \psi) = x$.
ψ *is called an* x*-expression of* ϕ.

Proof. We prove the contraposition. Let ϕ be an expression such that for all its sub-expressions of the form $\delta(\psi)$, there is no valuation μ such that $eval(\mu, \psi) = x$. Let now μ and μ' be two valuations that agree on $X \setminus \{x\}$. By structural induction on ϕ, $eval(\mu, \phi)$ (resp. $eval(\mu', \phi)$) does not depend on the value of $\mu(x)$ (resp. $\mu'(x)$). Hence, $eval(\mu, \phi) = eval(\mu', \phi)$ and we conclude that ϕ does not depend on x. □

Lemma 2. *If* ϕ *contains no nested* δ *operator, then* x *is not in the support of* $\psi = eval(\mu_{|\{x\}}, \phi)$ *for all valuations* μ *and addresses* x*. The converse is not true.*

Proof. We also prove this lemma by contraposition. Assume there exists a μ such that ψ has an x-expression ψ'. There exists an x-expression ϕ' of ϕ such that $eval(\mu_{|\{x\}}, \phi') = \psi'$. If ψ' were constant, then, according to definition 2, ψ' would be in D, and since it is an x-expression, ψ' would necessarily be equal to x. Thus, according to definition 2, $\delta(\phi')$ would be replaced by $\mu(\psi') = \mu(x)$ in ψ, so that ψ' would not be a sub-expression of ψ. This is contradictory, and proves that ψ' is not constant.
ψ' thus depends on at least an address $y \in X$, and, according to lemma 1, contains an occurrence of δ. It implies that ϕ' also contains an occurrence of δ, showing that ϕ contains nested δ operators.

Let $+$ denote any binary symbol in Σ. If $\phi = \delta(\delta(x) + \delta(y))$ and $\mu(x) + \mu(y) = x$, then $\delta(\mu(x) + \delta(y))$ still depends on x. This counter-example to the converse implication can be extended to a symbol of any arity $n \geq 2$, in case Σ contains no binary symbol. □

When there are nested δ operators, Lemma 2 states that substituting the content of an address x in ϕ, as section 4.2 naively does, may not completely remove the dependence on x. However, we can reduce this general case to the previous one by recursively solving nested x-expressions. This procedure terminates since each iteration strictly reduces the number of nested δ operators. This is discussed in section 4.3 and the correctness in section 4.4.

4.2 Without Nested δ Operators

The algorithm *EquivSplit* is shown in Algo. 1. It builds equivalence classes for $\sim_\phi^X$ dynamically based on successive substitution, refinement and merge steps on a partition of the input set. At step i:

- the substitution step uses the partition according to all possible contents of current address x_i (directly provided by the DDD encoding of valuations), to evaluate ϕ with each of these values;
- the refinement step refines the partition by recursively evaluating the reduced expressions over addresses $x_{i+1}, \ldots, x_{|X|}$;
- the merge step merges cells of the partition that lead to the same reduced expression over addresses $x_i, \ldots, x_{|X|}$.

At each step i, the goal becomes to remove any dependencies on x_i from the expression ϕ, allowing recursion over $x_{i+1}, \ldots, x_{|X|}$.

This algorithm is mutually recursive with *SolveSub* invoked on line 8. To help comprehension, we first consider the restricted case where no nested δ occur. In such a case, $SolveSub(\phi, V, i)$ always returns the singleton $\{(\phi, V)\}$. Hence, we study in this section the Algo. 1 independently from the algorithm of *SolveSub* presented in section 4.3.

From a programming language point of view, forbidding nested δ operators means that all addresses are known at compile time, and that no arithmetic on pointers occurs. By lemma 2, this restriction implies that if ϕ is an expression, x an address and μ a valuation, once ϕ is reduced with $\mu(x)$, it no longer depends on x.

Algorithm 1. EquivSplit(ϕ,V,i)

Input: ϕ an expression that does not depend on $x_1, \ldots, x_{i-1}$
Input: V a finite set of valuations
Input: i an integer between 1 and $|X|+1$
Output: a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that $c_1, \ldots, c_n$ are the equivalence classes of $\sim_\phi^{\{x_i, \ldots x_n\}}$ over V, and for each $1 \leq j \leq n$, $\phi_j = eval(\mu_{|\{x_i, \ldots x_n\}}, \phi)$ for any $\mu \in c_j$.

```
1  if φ is constant then
2      return {(V, φ)}
3  else
4      map < Expr, 2^V > res
5      let α_d = {μ ∈ V | μ(x_i) = d} for d ∈ D
6      foreach α_d ≠ ∅ do
           // Substitution
7          θ = φ[δ(x_i) ← d]
           // to remove nested δ operators
8          for (ψ, c) ∈ SolveSub(θ, α_d, i) do
               // Refinement
9              for (ψ', c') ∈ EquivSplit(ψ, c, i+1) do
                   // Merge
10                 res[ψ'] = res[ψ'] ∪ c'
11 return res
```

The base case of the recursion is when ϕ is constant, hence $\sim_\phi^{Y_i}$ has a single equivalence class V (lines 1-2). If $i = |X|+1$, by the precondition on the input ϕ, ϕ is constant.

The sets $(\alpha_d)_{d\in D}$ partition V into equivalence classes with respect to the value d of x_i (lines 5-6). Note that the symbolic encoding of valuations as DDD naturally provides this partition.

To each class α_d, we associate a reduced expression θ by replacing in ϕ variable x_i by its value d (line 7). Under our simplifying assumption, θ no longer depends on x_i, and $SolveSub(\theta, \alpha_d, i) = \{(\theta, \alpha_d)\}$. Thus, the loop on line 8 is reduced to a single call to line 9, that becomes: "$(\psi', c') \in EquivSplit(\theta, \alpha_d, i+1)$".

The loop on line 9 refines the partition element α_d (c in the general case) by recursively evaluating θ (ψ in the general case) on subsequent addresses $(x_{i+1}, \ldots, x_{|X|})$. Since elements from different α_d's may yield the same final value for ϕ, line 10 merges them into the final partition into equivalence classes for $\sim_\phi^{\{x_i,\ldots,x_{|X|}\}}$.

Invoking *EquivSplit*$(\phi, V, 0)$ returns the equivalence classes of elements in V with respect to $\sim_\phi^X$.

4.3 With Nested δ Operators

We now extend our algorithm to the general case. The precondition on the input ϕ for Algo. 1 is that ϕ does not depend on $x_1, \ldots, x_{i-1}$. Hence, recursion on line 9 requires that ψ does not depend on $x_1, \ldots, x_i$. The algorithm *SolveSub* addresses this problem by reducing x_i-expressions in θ by looking ahead the values of subsequent addresses $x_{i+1}, \ldots, x_{|X|}$. Lemma 2 shows that with no nested δ, looking zero addresses ahead suffices to eliminate the dependencies, and falls back to the case of 4.2. The algorithm *SolveSub* performs this reduction using the look-ahead, and returns a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that c_j are sets of valuations that agree on a look-ahead reduction ϕ_j of θ and that do not depend on x_i.

SolveSub computes in *res* a partition of V, and associates to each cell a simplified expression obtained by partially resolving ϕ, until all dependencies on x_i are removed. *tmp* is initialized as a single cell associated to ϕ (line 3). At each step of the while loop (line 4-5), an element (ψ, c) of *tmp* is treated. If the current expression ψ does not depend on x_i, the pair is moved to *res* (lines 11-12). Otherwise, we let θ be an x_i-expression of ψ (line 7). Recall, by lemma 1, that such a θ exists and has less nested δ operators than ψ. Any x-expression can be chosen and will lead to a correct result, hence the algorithm has some latitude at this point. Heuristically, to favor merging of partially resolved expressions, it is desirable to first treat x-expressions with a small co-domain (e.g. solve boolean sub-expressions first). Note that this is only possible with some additional knowledge of the signature's interpretation.

Recursion by invoking *EquivSplit* with θ (line 8) refines the cell c according to the value of θ. To each of these refined cells is associated the reduction of ψ obtained by substituting θ by its value (line 9). They are then added to *tmp* that merges the cells according to the reduced expression ψ' (line 11).

4.4 Correctness and Complexity

Sketch of the proof of correctness. We give here some intuition about the correctness of both algorithms.

Algorithm 2. SolveSub(ϕ,V,i)

Input: ϕ an expression that does not depend on $x_1, \ldots, x_{i-1}$
Input: V a set of valuations that all agree on the value d of x_i
Input: i an integer between 1 and $|X|$
Output: a set of pairs $\{(\phi_1, c_1), \ldots, (\phi_n, c_n)\}$ such that $c_1, \ldots, c_n$ is a partition of V, and for each $1 \leq j \leq n$, ϕ_j is a reduced expression obtained by removing all dependencies on x_i from ϕ, and all valuations in c_j agree on this reduction ϕ_j

```
1  map < Expr, 2^V > res
2  map < Expr, 2^V > tmp
3  tmp[φ] = V
4  while tmp is not empty do
5      (ψ, c) = tmp.pop()
6      if ψ has an x_i-expression then
7          θ = an x_i-expression of ψ
8          for (θ', c') ∈ EquivSplit(θ, c, i) do
9              ψ' = ψ[θ ← θ']
10             ψ' = ψ'[δ(x_i) ← d]
11             tmp[ψ'] = tmp[ψ'] ∪ c'
12     else
           // ψ does not depend on x_i
13         res[ψ] = res[ψ] ∪ c
14 return res
```

The recursive call on line 9 in *EquivSplit* at step i uses parameter $i+1$; since i is bounded by $|X|+1$ (height of the decision diagram), this recursion terminates. *SolveSub* recursively solves strict sub-expressions of ϕ, hence the recursion is bounded by the height of the syntactic tree. Since calls to *EquivSplit* from *SolveSub* always concern strictly smaller expressions, the mutual recursion is also bounded.

Both algorithms work by successively refining and coarsening a partition of the input set. Any time a pair (ψ, c) is inserted into the output, ψ is obtained by evaluating ϕ (or a derivative) on elements of c. Since the output is stored in a map, merging cells (ψ, c) and (ψ', c') respects the constraint that $\psi = \psi'$, hence c and c' belong to the same equivalence class.

Due to lack of space, the full proof is presented in a separate report [8].

Complexity of *EquivSplit*. In Algorithm 1, the α_d's for the loop on line 6 are already provided by the DDD representation of valuations, so that this loop is a just a walk of already computed sets. The main source of complexity in this function lies in the call to *SolveSub*. In the case when ϕ has no nested δ operators, then the loop on line 8 has a single pass. The recursion on line 9 explores the subsequent part of the DDD, so that, using a cache, the total complexity of *EquivSplit* is related to the size of the input DDD, rather than to the size of V.

Complexity of *SolveSub*. The look-ahead of *SolveSub*, performed on line 8 of function 2, refines the α_d in input. This refinement (that builds new decision diagrams) can

be arbitrarily fine, and depends on the input expression and the input set of valuations. The overall complexity of *SolveSub* is thus hard to predict and depends on the number of equivalence classes built.

A worst case for our technique would be an expression computing a hash value based on the values in all the memory slots. A perfect hash function would yield equivalence classes limited to singletons, hence encountering exponential worst case complexity (linear over states contained). Conversely, expressions with a small codomain (such as boolean expressions) give a small bound on the maximum number of equivalence classes manipulated by the algorithm. A peak effect for symbolic techniques occurs when an intermediate DDD size is proportional to its set size. This may occur anytime a partition element is built, hence finer partitions are more likely to induce a peak effect.

Caches. A cache for *EquivSplit* is built by associating to each ⟨DDD, expression⟩ pair the set {⟨DDD, expression value⟩} that partitions the input DDD into equivalence classes for the input expression. The full evaluation of various statements may thus share the cache allowing computation of common sub-expressions. Because it contains partial evaluations results, and no specific attempt is made to reconcile combined results, the structure of this cache differs from a decision diagram representing the full effects of transitions, although it allows to reconstruct the same transition information.

Variable Order. Much of the complexity for both of these algorithms depends on the variable ordering used in the DDD encoding. The equivalence classes depend on the order in which x_i's are visited. The representation size of the equivalence classes also strongly depends on this order. Heuristically, orderings that minimize invocations to *SolveSub* reduce the complexity. Limiting the depth of the look-ahead mechanism also helps to build DDD that share existing suffixes.

In our experiments, we adapted the FORCE algorithm [1]. Given a directed hypergraph where weighted edges represent constraints on variables (nodes of the hypergraph), FORCE heuristically computes an ordering on variables that minimizes the total weight. Expressions induce constraints on the variables in their support. By assigning a strong weight to constraints implying invocations to *SolveSub*, and small weight to constraints enforcing locality, we obtained satisfactory results.

4.5 Evaluating Assignments

We now informally present how to use our new algorithms to handle assignments of expressions to memory slots. We consider a semantic of a software system is described as sequences of assignments. An assignment is a pair of expressions (ϕ, ψ), where ϕ denotes the address of the affected memory slot and ψ the new value to assign to it. Allowing ϕ to depend on current memory state allows to model assignments such as `t[i] := 0`.

In our DDD implementation, an assignment is encoded as a homomorphism $\mathbb{D} \mapsto \mathbb{D}$. It evaluates both ϕ and ψ by walking the input DDD. As variables are encountered, ϕ and ψ are partially evaluated. If dependencies on current variable are not eliminated (nested δ), *SolveSub* is invoked. At some point, ϕ is reduced to a constant, which is the target of the assignment. When this target is reached, ψ must then be evaluated to a constant which may involve a look ahead using *SolveSub*.

Since our assignments are encoded as homomorphisms, they benefit from the automatic rewriting rules of [11]. These rules use the support of the expressions to skip don't-care variables and build clusters of transition effects. Our algorithm can also be implemented within other DD libraries. However, using DDD and homomorphisms allowed us to immediately benefit from these features.

5 Assessment

We compare our approach to related work and assess its efficiency compared to other symbolic techniques.

5.1 Related Work

To encode a transition, the original symbolic approach [5] relies on a second set of variables that associates to each variable its new state after the transition, for all potential states. The global transition relation is then the monolithic union (logical or of behaviours) of all possible transitions. This monolithic approach matches the synchronous semantics of hardware systems, but yields intractable representations in many cases.

This forced to introduce new strategies [14], where an explicitly managed set of DD store conjuncts of the transition relation. This process, called transition clustering, allows to overcome some of the limits of the monolithic approach.

For Globally Asynchronous Locally Synchronous (GALS) systems, [6], proposes to design the clustering according to the top-most variable in transition supports. The semantics of such systems is given as an asynchronous interleaving of locally synchronous actions (e.g. Petri nets). Such a clustering allows *saturation* to optimize the evaluation of the least fixpoint of a set of conjuncts: based on the interleaving semantics of the conjuncts, the fixpoint is first computed on lower parts of the DD.

A similar formalism is proposed by LTSmin [4]. A system is defined as consisting of k state variables with a discrete domain D and of transitions described primarily by their support composed of $k' \leq k$ variables. To compute the state space, LTSmin relies on third-party existing explicit model-checkers that provide a computation procedure called for each encountered value of the support in the global state space. Thanks to this projection, the number of these calls is bounded by $D^{k'}$ and in practice is limited to actually encountered states. This tool also implements state-of-the-art symbolic techniques, such as saturation, using classical encoding with two "before" and "after" variables per system state variable.

This approach is however severely challenged when the support grows. If the high-level model features array manipulation, pessimistic assumptions on the supports end up with supports including most (if not all) state variables. In such an extreme case, the explicit engine is invoked at least once for each state, negating any possible gain from the use of DD. Additionally, such individual insertion of paths in a DD is liable to produce exponential memory peak effects. Large supports also severely limit the possibilities of saturation as clusters are based on the support of transitions.

The algorithms we present in this paper partly overcome these difficulties. Large supports are often the result of array manipulation or composition of local effects induced by sequences of assignments. As we have seen, the support of an expression is

dynamically reduced. Large potential supports due to array manipulations are correctly resolved on-the-fly by *EquivSplit*. Compositions of effects are managed as explicit composition of homomorphisms, each of which has a support defined by its underlying expressions. Our fully symbolic encoding of the expressions avoids any explicit step where states are individually considered in the model-checking algorithm.

5.2 Implementation

To assess our algorithms, we chose to use benchmark models from the BEEM database [13], that are written in the Divine language [2]. To this end, we defined an intermediate formalism called Guarded-Action Language (GAL)[1], that can be manipulated symbolically with the algorithms described in this paper. This formalism defines a system's memory μ using integer variables and fixed size arrays of integers. Its transitions are composed of a guard that is a boolean expression over variables and a sequence of statements that are assignments of expressions to variables (or to cells of an array). A state of a GAL system is defined as the valuation of all variables. A transition is enabled in any state where the guard is true. Firing an enabled transition yields in a single step the successor state obtained by executing the assignments of the transition in an atomic sequence. The semantics are thus globally asynchronous, but sequences of statements are locally synchronous, reflecting the semantics of concurrent systems.

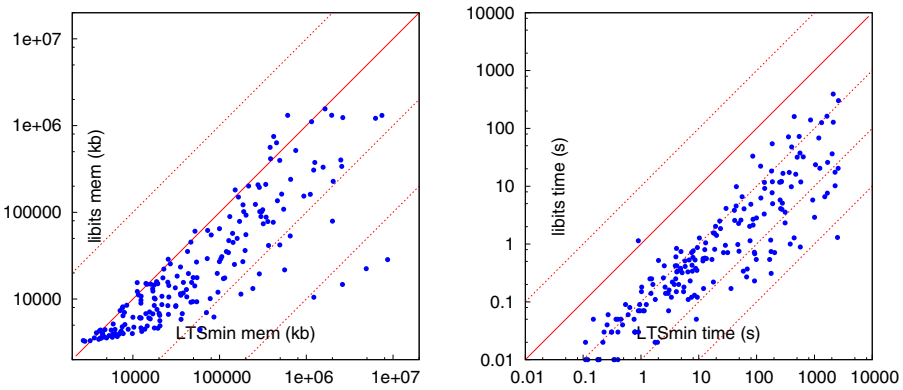
This small formalism offers a rich signature Σ consisting of all C operators for manipulation of the `int` data type and of arrays (including nested array expressions). There is no explicit support for pointers, though they can be simulated with an array *heap* and indexes into it. It also supports full C-like boolean expressions.

With these features, translation of Divine models into GAL was relatively straightforward. This technical work was done by adapting the code of LTSmin's wrapper for Divine models, where the semantic bridge to a system based on integer variables already existed. Divine is a language for describing processes that communicate through bounded channels, shared variables and/or synchronization. Channels are modeled using arrays. Synchronizations use a conjunction of local condition as a guard, and a sequence of local effects on each process as action. Priorities (deriving from the "commit" semantics of Divine) are enforced by adding the negation of the disjunction of the guards of higher priority to guards of transitions with lower priority.

5.3 Performance Assessment

To assess our new technique we built an extension to the `libits` tool[2], and compared its performance to classical state-of-the-art approaches, represented by the tools LTSmin [4], `super_prove` [3]. The performance comparison is based on the full set of models from the BEEM database [13]. Here we only report on reachability properties that were also provided in the context of a recent hardware model checking contest (HWMCC'12[3]) as SAT instances. Our implementation supports full CTL and LTL

[1] `http://move.lip6.fr/software/DDD/gal.php`
[2] `http://ddd.lip6.fr`
[3] `http://fmv.jku.at/hwmcc12`



(a) comparison on peak memory (b) comparison on time

Fig. 2. `libits` vs LTSmin, same variable ordering

model-checking of Divine models. All experiments were run on a Xeon 64 bits at 2.6 GHz processor.

LTSmin is a tool suite for model-checking, that implements state-of-the-art symbolic techniques (see 5.1). It can use several third-party DD libraries, but we configured it to use DDD to allow easier algorithmic comparison. Indeed the state encoding being provided by the same DDD library as ours, the main difference between this tool and ours is the use of *EquivSplit*.

`super_prove` is a SAT based model-checker. It was the winner of the "single safety/ bad-state property" track of the HWMCC'12, that contains the BEEM models. It is thus, to our knowledge, the best SAT-solver for this particular benchmark. SAT techniques are very different from those discussed in this paper, but raw performance comparisons on this benchmark are still possible.

`libits` is a DD-based verification library that uses both hierarchical set decision diagrams and DDD to support model-checking (CTL, LTL) of composition of labeled transition systems described symbolically. Transition systems can be described using several input formalisms, such as labeled discrete time Petri nets. The GAL formalism was embedded in this framework but only uses DDD.

Detailed results of experiments are presented as scatter plots comparing two tools over the whole benchmark. Each point represents a (model,formula) pair that was tested for reachability with both tools. A point below the diagonal means that `libits` is more efficient than the other tool. Our plots use a logarithmic scale. Lines parallel to the diagonal represent performance ratios of 10, 100 ... (resp. 0.1, 0.01 ...).

Table 1. `libits` vs LTSmin

models tested	treated by libits	treated by LTSmin	treated by both	treated by none
293	264	212	197	22

libits vs. LTSmin. We compare the performance for the generation of the state space of the models, with 1 hour and 10Gb containment. Statistics of the results are shown in Fig. 1, and more detailed results are shown on Fig. 2. The results confirm that our *EquivSplit* algorithm performs better than the classical symbolic approach. With the same implementation of DDD and the same variable ordering, our implementation is up to 1000 times faster and 100 times less memory consuming than LTSmin. For a dozen models, LTSmin is slightly more memory efficient than libits, but this can be attributed to side-effects of the garbage collection policy.

libits vs. super_prove. We compare the performance of both `libits` and `super_prove`, with a containment of 1Gb in memory and 900 seconds wall clock time (`super_prove` uses 4 cores while `libits` is mono-threaded). These are the containment settings used in the HWMCC competition. Summary of the results are shown in Fig 4, and detailed results are presented on Fig. 3. We only compared the time usage, since the memory consumption for SAT techniques is usually insignificant. Note that all `libits`'s fails are due to a memory overflow, whereas all `super_prove`'s fails are due to a time overflow.

`libits` treats about 35% more models than `super_prove`. Also, `libits` is quicker than `super_prove` for 80% of the models treated by both tools, with a speed-up factor up to 1000. On the other models, `super_prove`'s speed-up factor ranges up to 100.

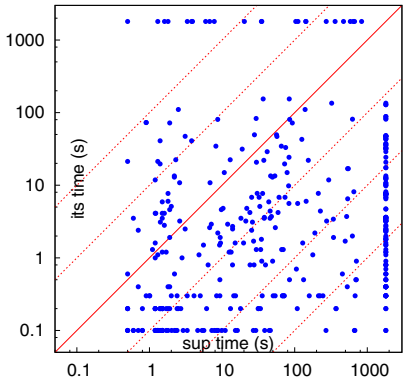


Fig. 3. Time comparison between `libits` and `super_prove`

	# unsat # unsat	mean time unsat (s)	# sat # sat	mean time sat (s)
libits	184	14.6	192	8.6
super_prove	112	140.6	170	45.1

models tested	treated by libits	treated by super_prove	treated by both	treated by none
456	376	282	258	56

Fig. 4. `libits` vs `super_prove` (top: mean runtime and bottom: number of instances solved)

The top table in Fig. 4 shows that `libits` runs on average 5 times faster on satisfied properties and 10 times faster on unsatisfied properties than `super_prove` that stops as soon as it finds a solution for satisfied instances. Our tool regularly interrupts the computation to check whether a solution exists in the states computed so far. When these checks are deactivated, `libits` is 4 times faster on satisfied properties and 14 times faster on unsatisfied properties. Unsatisfied instances require both tools to explore the whole reachability graph: these are the hardest problems.

On this benchmark, we show that state-of-the-art symbolic manipulation of decision diagrams can still outperform the best SAT-based techniques.

6 Conclusion

This paper proposes a new algorithm, *EquivSplit*, that allows more efficient symbolic manipulation of software-like models. It uses equivalence relations to avoid explicit manipulation of states. Assessment on a large third-party benchmark shows that this approach improves existing decision diagram-based techniques, and can outperform SAT-based ones.

Our algorithm supports arbitrary signatures (languages), and can be used with any type of decision diagrams. It uses information provided by the high-level expressions of the transition relation to dynamically optimize computations.

Using the *EquivSplit* algorithm, we are currently investigating the combination of symmetries with decision diagrams as an extension of previous work performed without this contribution [9].

References

1. Aloul, F., Markov, I., Sakallah, K.: Force: a fast and easy-to-implement variable-ordering heuristic. In: 13th ACM Great Lakes symposium on VLSI, pp. 116–119. ACM (2003)
2. Barnat, J., Brim, L., Češka, M., Ročkai, P.: DiVinE: Parallel Distributed Model Checker (Tool paper). In: Parallel and Distributed Methods in Verification and High Performance Computational Systems Biology (HiBi/PDMC), pp. 4–7. IEEE (2010)
3. Berkeley Logic Synthesis and Verification Group: ABC: A System for Sequential Synthesis and Verification, `http://www.eecs.berkeley.edu/~alanmi/abc/` (release October 12, 2006)
4. Blom, S., van de Pol, J., Weber, M.: LTSMIN: distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
5. Burch, J., Clarke, E., et al.: Symbolic model checking: 10^{20} States and beyond. Information and Computation 98(2), 142–170 (1992)
6. Ciardo, G., Marmorstein, R., Siminiceanu, R.: Tools and Algorithms for the Construction and Analysis of Systems, pp. 379–393 (2003)
7. Clarke, E., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
8. Colange, M., Baarir, S., Kordon, F., Thierry-Mieg, Y.: Towards Distributed Software Model-Checking using Decision Diagrams (extended version with annexes). Tech. rep., CoRR (2013), `http://arxiv.org/find/all/1/all:+kordon`
9. Colange, M., Kordon, F., Thierry-Mieg, Y., Baarir, S.: State Space Analysis using Symmetries on Decision Diagrams. In: 12th International Conference on Application of Concurrency to System Design (ACSD), pp. 164–172. IEEE Computer Society (June, 2012)
10. Couvreur, J.-M., Encrenaz, E., Paviot-Adet, E., Poitrenaud, D., Wacrenier, P.-A.: Data decision diagrams for petri net analysis. In: Esparza, J., Lakos, C.A. (eds.) ICATPN 2002. LNCS, vol. 2360, pp. 101–120. Springer, Heidelberg (2002)
11. Hamez, A., Thierry-Mieg, Y., Kordon, F.: Hierarchical set decision diagrams and automatic saturation. In: van Hee, K.M., Valk, R. (eds.) PETRI NETS 2008. LNCS, vol. 5062, pp. 211–230. Springer, Heidelberg (2008)
12. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering 23, 279–295 (1997)
13. Pelánek, R.: Beem: Benchmarks for explicit model checkers. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 263–267. Springer, Heidelberg (2007)
14. Ranjan, R., Aziz, A., Brayton, R., Plessier, B., Pixley, C.: Efficient bdd algorithms for fsm synthesis and verification. In: IWLS 1995, Lake Tahoe, CA, vol. 253, p. 254 (1995)

Automatic Abstraction in SMT-Based Unbounded Software Model Checking*

Anvesh Komuravelli, Arie Gurfinkel, Sagar Chaki, and Edmund M. Clarke

Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. Software model checkers based on under-approximations and SMT solvers are very successful at verifying safety (*i.e.*, reachability) properties. They combine two key ideas – (a) *concreteness*: a counterexample in an under-approximation is a counterexample in the original program as well, and (b) *generalization*: a proof of safety of an under-approximation, produced by an SMT solver, are generalizable to proofs of safety of the original program. In this paper, we present a combination of *automatic abstraction* with the under-approximation-driven framework. We explore two iterative approaches for obtaining and refining abstractions – *proof based* and *counterexample based* – and show how they can be combined into a unified algorithm. To the best of our knowledge, this is the first application of Proof-Based Abstraction, primarily used to verify hardware, to Software Verification. We have implemented a prototype of the framework using Z3, and evaluate it on many benchmarks from the Software Verification Competition. We show experimentally that our combination is quite effective on hard instances.

1 Introduction

Algorithms based on generalizing from under-approximations are very successful at verifying safety properties, *i.e.*, absence of bad executions (e.g., [2,10,26]). Those techniques use what we call a *Bounded Model Checking-Based Model Checking* (2BMC). The key idea of 2BMC is to iteratively construct an under-approximation U of the target program P by unwinding its transition relation and check whether U is safe using Bounded Model Checking (BMC) [8]. If U is unsafe, so is P. Otherwise, a proof π_U is produced explaining *why* U is safe. Finally, π_U is generalized (if possible) to a safety proof of P. Notable instances of

* This research was sponsored by the National Science Foundation grants no. DMS1068829, CNS0926181 and CNS0931985, the GSRC under contract no. 1041377, the Semiconductor Research Corporation under contract no. 2005TJ1366, the Office of Naval Research under award no. N000141010188 and the CMU-Portugal Program. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material has been approved for public release and unlimited distribution. (DM-0000279).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 846–862, 2013.

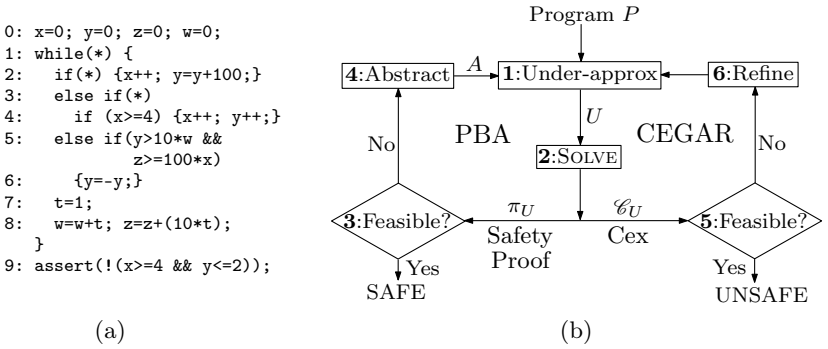


Fig. 1. (a) A program P_g by Gulavani et al. [18]; (b) an overview of SPACER.

2BMC are based on interpolation (e.g., [2,26]) or Property Directed Reachability (PDR) [9,13] (e.g., [10,21]).

At the same time, automatic abstraction refinement, such as CounterExample Guided Abstraction Refinement (CEGAR) [11], is very effective [2,7,20]. The idea is to iteratively construct, verify, and refine an abstraction (*i.e.*, an over-approximation) of P based on abstract counterexamples. In this paper, we present SPACER[1], an algorithm that combines abstraction with 2BMC.

For example, consider the safe program P_g by Gulavani et al. [18] shown in Fig. 1(a). P_g is hard for existing 2BMC techniques. For example, μZ engine of Z3 [12] (v4.3.1) that implements Generalized PDR [21] cannot solve it within an hour. However, its abstraction $\hat{P}_g$ obtained by replacing line 7 with a non-deterministic assignment to `t` is solved by the same engine in under a second. Our implementation of SPACER finds a safe abstraction of P_g in under a minute (the transition relation of the abstraction we automatically computed is a non-trivial generalization of that of P_g and does not correspond to $\hat{P}_g$).

SPACER tightly connects *proof-based* (PBA) and *counterexample-based* (CEGAR) abstraction-refinement schemes. An overview of SPACER is shown in Fig. 1(b). The input is a program P with a designated error location *er* and the output is either SAFE with a proof that *er* is unreachable, or UNSAFE with a counterexample to *er*. SPACER is sound, but obviously incomplete, *i.e.*, it is not guaranteed to terminate.

During execution, SPACER maintains an abstraction A of P, and an under-approximation U of A. We require that the safety problem for U is decidable. So, U is obtained by considering finitely many finitary executions of A. Initially, A is any abstraction of P (or P itself) and U is some under-approximation (step 1) of A. In each iteration, the main decision engine, called SOLVE, takes U and outputs either a proof π_U of safety (as an inductive invariant) or a counterexample trace $\mathscr{C}_U$ of U (step 2). In practice, SOLVE is implemented by an interpolating SMT-solver (e.g., [17,23]), or a generalized Horn Clause solver (e.g., [28,16,21]). If U is safe and π_U is also valid for P (step 3), SPACER terminates with SAFE;

[1] Software Proof-based Abstraction with CounterExample-based Refinement.

otherwise, it constructs a new abstraction $\hat{A}$ (step 4) using π_U, picks an under-approximation $\hat{U}$ of $\hat{A}$ (step 1), and goes into the next iteration. If U is unsafe and $\mathscr{C}_U$ is a feasible trace of P (step 5), Spacer terminates with UNSAFE; otherwise, it refines the under-approximation U to refute $\mathscr{C}_U$ (step 6) and goes to the next iteration. Spacer is described in Section 4 and a detailed run of the algorithm on an example is given in Section 2.

Note that the left iteration of Spacer (steps 1, 2, 3, 4) is PBA: in each iteration, an under-approximation is solved, a new abstraction based on the proof is computed and a new under-approximation is constructed. To the best of our knowledge, this is the first application of PBA to Software Model Checking. The right iteration (steps 1, 2, 5, 6) is CEGAR: in each iteration, (an under-approximation of) an abstraction is solved and refined by eliminating spurious counterexamples. Spacer exploits the natural duality between the two.

While Spacer is not complete, each iteration makes progress either by proving safety of a bigger under-approximation, or by refuting a spurious counterexample. Thus, when resources are exhausted, Spacer can provide useful information for other verification attempts and for increasing confidence in the program.

We have implemented Spacer using μZ [21] as Solve (Section 5) and evaluated it on many benchmarks from the 2nd Software Verification Competition[2] (SV-COMP'13). Our experimental results (see Section 6) show that the combination of 2BMC and abstraction outperforms 2BMC on hard benchmarks.

In summary, the paper makes the following contributions: (a) an algorithm, Spacer, that combines abstraction and 2BMC and tightly connects proof- and counterexample-based abstractions, (b) an implementation of Spacer using μZ engine of Z3 and (c) experimental results showing the effectiveness of Spacer.

2 Overview

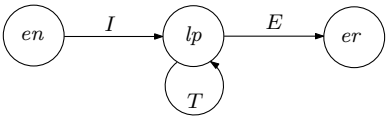
In this section, we illustrate Spacer on the program P shown in Fig. 2(a). Function `nd()` returns a value non-deterministically and `assume(0)` aborts an execution. Thus, at least one of the updates on lines 3, 4 and 5 must take place in every iteration of the loop on line 2. Note that the variable `c` counts down the number of iterations of the loop to 0, upper bounded by `b`. A restriction to `b` is an under-approximation of P. For example, adding '`assume(b<=0);`' to line 1 corresponds to the under-approximation of P that allows only loop-free executions; adding '`assume(b<=1);`' to line 1 corresponds to the under-approximation that allows at most one execution through the loop, etc. While in this example the *counter variable* `c` is part of P, we synthesize such variables automatically in practice (see Section 5).

Semantically, P is given by the transition system shown in Fig 2(b). The control locations *en*, *lp*, and *er* correspond to lines 0, 2, and 8 in P, respectively. An edge from ℓ_1 to ℓ_2 corresponds to all loop-free executions starting at ℓ_1 and ending at ℓ_2. For example, the self-loop on *lp* corresponds to the body of the loop. Finally, every edge is labeled by a formula over current (unprimed)

[2] `http://sv-comp.sosy-lab.org`

```
0: x=0; y=0; z=0; w=0; c=nd();
1: b=nd();
2: while (0<c<=b) {
3:   if (nd()) {x++; y=y+100;}
4:   else if (nd() && x>=4) {x++; y++;}
5:   else if (y>10*w && z>=100*x) {y=-y;}
6:   else assume (0);
7:   w++; z=z+10; c--;
   }
8: assert (!(c==0 && x>=4 && y<=2));
```

(a)



$$I \equiv (x' = y' = z' = w' = 0)$$
$$T \equiv [(x' = x + 1 \ \wedge \ y' = y + 100) \vee (x \geq 4 \ \wedge \ x' = x + 1 \ \wedge \ y' = y + 1) \vee (y > 10w \ \wedge \ z \geq 100x \ \wedge \ y' = -y \ \wedge \ x' = x)] \wedge w' = w + 1 \ \wedge \ z' = z + 10 \ \wedge \ 0 < c \leq b \ \wedge \ c' = c - 1$$
$$E \equiv c = 0 \ \wedge \ x \geq 4 \ \wedge \ y \leq 2$$

(b)

Fig. 2. (a) A program P and (b) its transition system

$$\hat{I}_1 \equiv (x' = y' = z' = w' = 0)$$
$$\hat{T}_1 \equiv [(x' = x + 1) \vee (x \geq 4 \ \wedge \ x' = x + 1) \vee (y > 10w \ \wedge \ z \geq 100x)] \wedge 0 < c \leq b \ \wedge \ c' = c - 1$$
$$\hat{E}_1 \equiv c = 0 \ \wedge \ x \geq 4$$

(a) $\hat{P}_1$

$$\hat{I}_2 \equiv (x' = y' = z' = w' = 0)$$
$$\hat{T}_2 \equiv [(x' = x + 1 \ \wedge \ y' = y + 100) \vee (x \geq 4 \ \wedge \ x' = x + 1 \ \wedge \ y' = y + 1) \vee (y > 10w \ \wedge \ z \geq 100x)] \wedge 0 < c \leq b \ \wedge \ c' = c - 1$$
$$\hat{E}_2 \equiv c = 0 \ \wedge \ x \geq 4 \ \wedge \ y \leq 2$$

(b) $\hat{P}_2$

Fig. 3. Abstractions $\hat{P}_1$ and $\hat{P}_2$ of P in Fig. 2(b)

and next-state (primed) variables denoting the semantics of the corresponding executions. Hence, I and E denote the initial and error conditions, respectively, and T denotes the loop body. In the rest of the paper, we do not distinguish between semantic and syntactic representations of programs.

Our goal is to find a safety proof for P, *i.e.*, a labeling π of *en*, *lp* and *er* with a set of formulas (called *lemmas*) that satisfies safety, initiation and inductiveness:

$$\bigwedge \pi(er) \Rightarrow \bot, \quad \top \Rightarrow \bigwedge \pi(en), \quad \forall \ell_1, \ell_2 \,.\, \left(\bigwedge \pi(\ell_1) \ \wedge \ \tau(\ell_1, \ell_2)\right) \Rightarrow \bigwedge \pi(\ell_2)'.$$

where $\tau(\ell_1, \ell_2)$ is the label of edge from ℓ_1 to ℓ_2, and for an expression X, X' is obtained from X by priming all variables. In the following, we refer to Fig. 1(b) for the steps of the algorithm.

Steps 1 and 2. Let U_1 be the under-approximation obtained from P by conjoining $(b \leq 2)$ to T. It is safe, and suppose that SOLVE returns the safety proof π_1, shown in Fig. 4(a).

Step 3. To check whether π_1 is also a safety proof of the concrete program P, we extract a *Maximal Inductive Subset* (MIS), $\mathcal{I}_1$ (shown in Fig. 4(b)), of π_1, with respect to P. That is, for every location ℓ, $\mathcal{I}_1(\ell) \subseteq \pi_1(\ell)$, and $\mathcal{I}_1$ satisfies the initiation and inductiveness conditions above. $\mathcal{I}_1$ is an inductive invariant

$en : \{\}$
$lp : \{(z \leq 100x - 90 \vee y \leq 10w), z \leq 100x, x \leq 2, (x \leq 0 \vee c \leq 1), (x \leq 1 \vee c \leq 0)\}$
$er : \{\bot\}$

(a) π_1: safety proof of U_1.

$en : \{\}$
$lp : \{(z \leq 100x - 90 \vee y \leq 10w), z \leq 100x\}$
$er : \{\}$

(b) $\mathcal{I}_1$: invariants of P.

$en : \{\}$
$lp : \{(z \leq 100x - 90 \vee y \leq 10w), z \leq 100x, y \geq 0, (x \leq 0 \vee y \geq 100)\}$
$er : \{\bot\}$

(c) π_3: safety proof of U_3.

Fig. 4. Proofs and invariants for the running example in Section 2

of P, but is not safe (er is not labeled with $\bot$). Hence, π_1 does not contain a feasible proof, and another iteration of SPACER is required.

Step 4. We obtain an abstraction $\hat{P}_1$ of P for which, assuming the invariants in $\mathcal{I}_1$, π_1 is a safety proof for the first two iterations of the loop (*i.e.*, when $b \leq 2$). For this example, let $\hat{P}_1$ be as shown in Fig. 3(b). Note that $\hat{T}_1$ has no constraints on the next-state values of z, y and w. This is okay for π_1 as $\mathcal{I}_1(lp)$ already captures the necessary relation between these variables. In other words, while $\hat{T}_1$ is a structural (or *syntactic*) abstraction [6], we consider its restriction to the invariants $\mathcal{I}_1$ making it a more expressive, *semantic* abstraction. The next iteration of SPACER is described below.

Steps 1 and 2. Let U_2 be the under-approximation obtained from $\hat{P}_1$ by conjoining $(b \leq 4) \wedge \mathcal{I}_1 \wedge \mathcal{I}_1'$ to $\hat{T}_1$. It is not safe and let SOLVE return a counterexample $\mathscr{C}_2$ as the pair $\langle \bar{\ell}, \bar{s} \rangle$ of the following sequences of locations and states, corresponding to incrementing x from 0 to 4 with an unconstrained y:

$$\begin{aligned} \bar{\ell} &\equiv \langle en, lp, lp, lp, lp, lp, er \rangle \\ \bar{s} &\equiv \langle (0,0,0,0,0,0), (0,0,0,0,4,4), (1,0,0,0,3,4), (2,0,0,0,2,4), \\ &\quad (3,0,0,0,1,4), (4,3,0,0,0,4), (4,3,0,0,0,4) \rangle \end{aligned} \tag{1}$$

where a state is a valuation to the tuple (x, y, z, w, c, b).

Steps 5 and 6. $\mathscr{C}_2$ is infeasible in P as the last state does not satisfy E. $\hat{P}_1$ is refined to $\hat{P}_2$, say as shown in Fig. 3(b), by adding the missing constraints on y.

Steps 1 and 2. Let U_3 be the under-approximation obtained from $\hat{P}_2$ by conjoining $(b \leq 4) \wedge \mathcal{I}_1 \wedge \mathcal{I}_1'$ to $\hat{T}_2$. It is safe, and let SOLVE return the proof π_3 shown in Fig. 4(c).

Step 3. π_3 is a MIS of itself, with respect to P. Thus, it is a safety proof for P and SPACER terminates.

While we have carefully chosen the under-approximations to save space, the abstractions, lemmas and invariants shown above were all computed automatically by our prototype implementation starting with the initial under-approximation of $b \leq 0$ and incrementing the upper bound by 1, each iteration. Even on this small example, our prototype, built using μZ, is five times faster than μZ by itself.

3 Preliminaries

This section defines the terms and notation used in the rest of the paper.

Definition 1 (Program). *A program P is a tuple $\langle L, \ell^o, \ell^e, V, \tau\rangle$ where*

1. *L is the set of control locations,*
2. *$\ell^o \in L$ and $\ell^e \in L$ are the unique initial and error locations,*
3. *V is the set of all program variables (Boolean or Rational), and*
4. *$\tau : L \times L \to BExpr(V \cup V')$ is a map from pairs of locations to Boolean expressions over $V \cup V'$ in propositional Linear Rational Arithmetic.*

Intuitively, $\tau(\ell_i, \ell_j)$ is the relation between the current values of V at ℓ_i and the next values of V at ℓ_j on a transition from ℓ_i to ℓ_j. We refer to τ as the transition relation. Without loss of generality, we assume that $\forall \ell \in L \centerdot \tau(\ell, \ell^o) = \bot \wedge \tau(\ell^e, \ell) = \bot$. We refer to the components of P by a subscript, e.g., L_P.

Fig. 2(b) shows an example program with $L = \{en, lp, er\}$, $\ell^o = en$, $\ell^e = er$, $V = \{x, y, z, w, c, b\}$, $\tau(en, lp) = I$, $\tau(lp, lp) = T$, $\tau(lp, er) = E$.

Let $P = \langle L, \ell^o, \ell^e, V, \tau\rangle$ be a program. A *control path* of P is a finite[3] sequence of control locations $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$, beginning with the initial location ℓ^o, such that $\tau(\ell_i, \ell_{i+1}) \neq \bot$ for $0 \leq i < k$. A *state* of P is a valuation to all the variables in V. A control path $\langle \ell^o = \ell_0, \ell_1, \ldots, \ell_k\rangle$ is called *feasible* iff there is a sequence of states $\langle s_0, s_1, \ldots, s_k\rangle$ such that

$$\forall 0 \leq i < k \centerdot \tau(\ell_i, \ell_{i+1})[V \leftarrow s_i, V' \leftarrow s_{i+1}] = \top \tag{2}$$

i.e., each successive and corresponding pair of locations and states satisfy τ.

For example, $\langle en, lp, lp, lp\rangle$ is a feasible control path of the program in Fig. 2(b) as the sequence of states $\langle(0,0,0,0,0,0), (0,0,0,0,2,2), (1,100,1,10, 1,2), (2,200,2,20,0,2)\rangle$ satisfies (2).

A location ℓ is *reachable* iff there exists a feasible control path ending with ℓ. P is *safe* iff ℓ^e is *not* reachable. For example, the program in Fig. 2(b) is safe. P is *decidable*, when the safety problem of P is decidable. For example, the program U obtained from P in Fig. 2(b) by replacing b with 5 is decidable because (a) U has finitely many feasible control paths, each of finite length and (b) Linear Arithmetic is decidable.

Definition 2 (Safety Proof). *A* safety proof *for P is a map $\pi : L \to 2^{BExpr(V)}$ such that π is safe and inductive,* i.e.*,*

$$\bigwedge \pi(\ell^e) \Rightarrow \bot, \quad \top \Rightarrow \bigwedge \pi(\ell^o), \quad \forall \ell_i, \ell_j \in L \centerdot \Big(\bigwedge \pi(\ell_i) \ \wedge \ \tau(\ell_i, \ell_j)\Big) \Rightarrow \bigwedge \pi(\ell_j)'.$$

For example, Fig. 4(c) shows a safety proof for the program in Fig. 2(b). Note that whenever P has a safety proof, P is safe.

A *counterexample to safety* is a pair $\langle \bar{\ell}, \bar{s}\rangle$ such that $\bar{\ell}$ is a feasible control path in P ending with ℓ^e and $\bar{s}$ is a corresponding sequence of states satisfying τ along $\bar{\ell}$. For example, $\hat{P}_2$ in Fig. 3(b) admits the counterexample $\mathscr{C}_2$ shown in (1) in Section 2.

[3] In this paper, we deal with safety properties only.

Definition 3 (Abstraction Relation). *Given two programs,* $P_1 = \langle L_1, \ell_1^o, \ell_1^e, V_1, \tau_1 \rangle$ *and* $P_2 = \langle L_2, \ell_2^o, \ell_2^e, V_2, \tau_2 \rangle$, P_2 *is an* abstraction (i.e., *an over-approximation*) *of* P_1 *via a surjection* $\sigma : L_1 \to L_2$, *denoted* $P_1 \preceq_\sigma P_2$, *iff*

$$V_1 = V_2, \quad \sigma(\ell_1^o) = \ell_2^o, \quad \sigma(\ell_1^e) = \ell_2^e, \quad \forall \ell_i, \ell_j \in L_1 \,.\, \tau_1(\ell_i, \ell_j) \Rightarrow \tau_2(\sigma(\ell_i), \sigma(\ell_j)).$$

P_1 *is called a* refinement (i.e., *an under-approximation*) *of* P_2. *We say that* P_2 *strictly abstracts* P_1 *via* σ, *denoted* $P_1 \prec_\sigma P_2$, *iff* $(P_1 \preceq_\sigma P_2) \wedge \neg\exists \nu \,.\, (P_2 \preceq_\nu P_1)$. *When* σ *is not important, we drop the subscript.*

That is, P_2 abstracts P_1 iff there is a surjective map σ from L_1 to L_2 such that every feasible transition of P_1 corresponds (via σ) to a feasible transition of P_2. For example, if P_1 is a finite unrolling of P_2, then σ maps the locations of P_1 to the corresponding ones in P_2. P_2 *strictly abstracts* P_1 iff $P_1 \preceq P_2$ and there is no surjection ν for which $P_2 \preceq_\nu P_1$. For example, $P \prec_{id} \hat{P}_1$, where P is in Fig. 2(b) and $\hat{P}_1$ is in Fig. 3(a).

We extend $\sigma : L_1 \to L_2$ from locations to control paths in the straightforward way. For a counterexample $\mathscr{C} = \langle \bar{\ell}, \bar{s} \rangle$, we define $\sigma(\mathscr{C}) \equiv \langle \sigma(\bar{\ell}), \bar{s} \rangle$. For a transition relation τ on L_2, we write $\sigma(\tau)$ to denote an embedding of τ via σ, defined as follows: $\sigma(\tau)(\ell_1, \ell_2) = \tau(\sigma(\ell_1), \sigma(\ell_2))$. For example, in the definition above, if $P_1 \preceq_\sigma P_2$, then $\tau_1 \Rightarrow \sigma(\tau_2)$.

4 The Algorithm

In this section, we describe Spacer at a high-level. Low-level details of our implementation are described in Section 5. The pseudo-code of Spacer is shown in Fig. 5. The top level routine Spacer decides whether an input program P (passed through the global variable) is safe. It maintains (a) invariants $\mathcal{I}$ such that $\mathcal{I}(\ell)$ is a set of constraints satisfied by all the reachable states at location ℓ of P (b) an abstraction A of P, (c) a decidable under-approximation U of A and (d) a surjection σ such that $U \preceq_\sigma A$. Spacer ensures that $P \preceq_{id} A$, *i.e.*, A differs from P only in its transition relation. Let $A_\mathcal{I}$ denote the restriction of A to the invariants in $\mathcal{I}$ by strengthening τ_A to $\lambda \ell_1, \ell_2 . \mathcal{I}(\ell_1) \wedge \tau_A(\ell_1, \ell_2) \wedge \mathcal{I}(\ell_2)'$. Similarly, let $U_\mathcal{I}$ denote the strengthening of τ_U to $\lambda \ell_1, \ell_2 . \mathcal{I}(\sigma(\ell_1)) \wedge \tau_U(\ell_1, \ell_2) \wedge \mathcal{I}(\sigma(\ell_2))'$. Spacer assumes the existence of an oracle, Solve, that decides whether $U_\mathcal{I}$ is safe and returns either a safety proof or a counterexample.

Spacer initializes A to P and $\mathcal{I}$ to the empty map (line 1), calls InitU(A) to initialize U and σ (line 2) and enters the main loop (line 3). In each iteration, safety of $U_\mathcal{I}$ is checked with Solve (line 4). If $U_\mathcal{I}$ is safe, the safety proof π is checked for feasibility w.r.t. the original program P, as follows. First, π is mined for new invariants of P using ExtractInvs (line 6). Then, if the invariants at ℓ_P^e are unsatisfiable (line 7), the error location is unreachable and Spacer returns Safe (line 8). Otherwise, A is updated to a new proof-based abstraction via Abstract (line 9), and a new under-approximation is constructed using NextU (line 10). If, on the other hand, $U_\mathcal{I}$ is unsafe at line 4, the counterexample $\mathscr{C}$ is validated using Refine (line 11). If $\mathscr{C}$ is feasible, Spacer returns Unsafe (line 13), otherwise, both A and U are refined (lines 20 and 21).

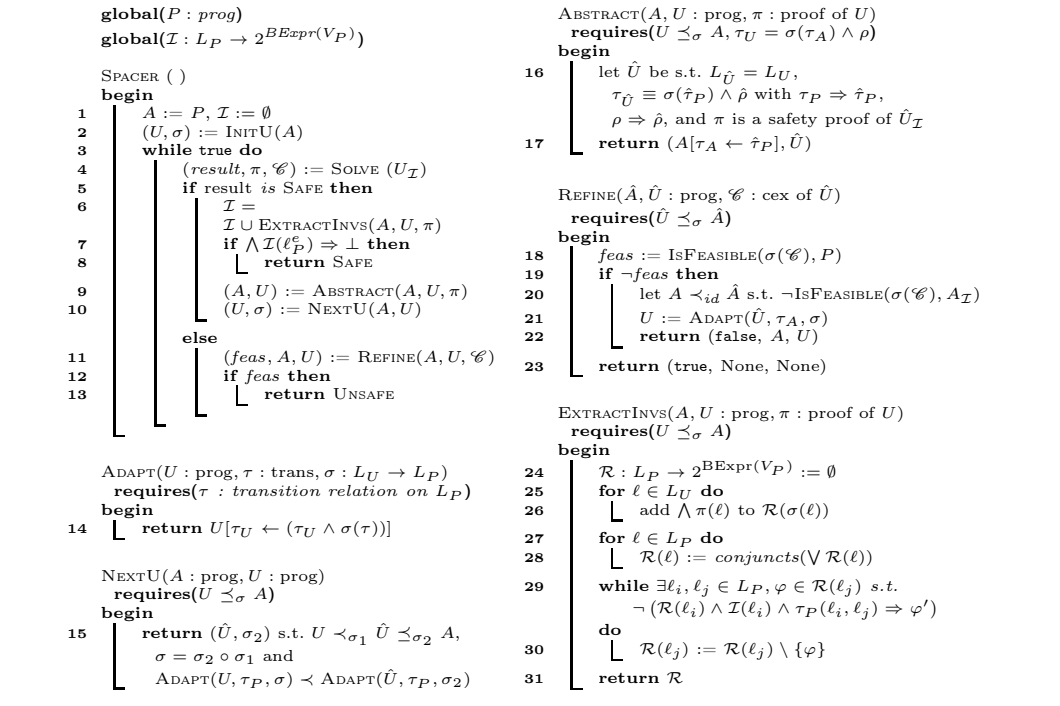


Fig. 5. Pseudo-code of SPACER

Next, we describe these routines in detail. Throughout, fix U, σ and A such that $U \preceq_\sigma A$.

EXTRACTINVS. For every $\ell \in L$, the lemmas of all locations in L_U which map to ℓ, via the surjection $\sigma : L_U \to L_P (= L_A)$, are first collected into $\mathcal{R}(\ell)$ (lines 25–26). The disjunction of $\mathcal{R}(\ell)$ is then broken down into conjuncts and stored back in $\mathcal{R}(\ell)$ (lines 27–28). For e.g., if $\mathcal{R}(\ell) = \{\phi_1, \phi_2\}$, obtain $\phi_1 \vee \phi_2 \equiv \bigwedge_j \psi_j$ and update $\mathcal{R}(\ell)$ to $\{\psi_j\}_j$. Then, the invariants are extracted as the maximal subset of $\mathcal{R}(\ell)$ that is mutually inductive, relative to $\mathcal{I}$, w.r.t. the concrete transition relation τ_P. This step uses the iterative algorithm on lines 29–30 and is similar to HOUDINI [15].

ABSTRACT first constructs an abstraction $\hat{U}$ of U, such that π is a safety proof for $\hat{U}_\mathcal{I}$ and then, uses the transition relation of $\hat{U}$ to get the new abstraction. W.l.o.g., assume that τ_U is of the form $\sigma(\tau_A) \wedge \rho$. That is, τ_U is an embedding of τ_A via σ strengthened with ρ. An abstraction $\hat{U}$ of U is constructed such that $\tau_{\hat{U}} = \sigma(\hat{\tau}_P) \wedge \hat{\rho}$, where $\hat{\tau}_P$ abstracts the concrete transition relation τ_P, $\hat{\rho}$ abstracts ρ and π proves $\hat{U}_\mathcal{I}$ (line 16). The new abstraction is then obtained from A by replacing the transition relation by $\hat{\tau}_P$ (line 17).

NEXTU returns the next under-approximation $\hat{U}$ to be solved. It ensures that $U \prec \hat{U}$ (line 15), and that the surjections between U, $\hat{U}$ and A compose so

that the corresponding transitions in U and $\hat{U}$ map to the same transitions of the common abstraction A. Furthermore, to ensure progress, NextU ensures that $\hat{U}$ contains *more concrete* behaviors than U (the last condition on line 15). The helper routine Adapt strengthens the transition relation of an under-approximation by an embedding (line 14).

Refine checks if the counterexample $\mathscr{C}$, via σ, is feasible in the original program P using IsFeasible (line 18). If $\mathscr{C}$ is feasible, Refine returns saying so (line 23). Otherwise, $\hat{A}$ is refined to A to (at least) eliminate $\mathscr{C}$ (line 20). Thus, $A \prec_{id} \hat{A}$. Finally, $\hat{U}$ is strengthened with the refined transition relation via Adapt (line 21).

The following statements show that Spacer is sound and maintains progress.

Lemma 1 (Inductive Invariants). *In every iteration of* Spacer, $\mathcal{I}$ *is inductive with respect to* τ_P.

Theorem 1 (Soundness). P *is safe* (*unsafe*) *if* Spacer *returns* Safe (Unsafe).

Theorem 2 (Progress). *Let* A_i, U_i, *and* $\mathscr{C}_i$ *be the values of* A, U, *and* $\mathscr{C}$ *in the* i^{th} *iteration of* Spacer *with* $U_i \preceq_{\sigma_i} A_i$ *and let* $\dot{U}_i$ *denote the concretization of* U_i, i.e., *result of* Adapt(U_i, τ_P, σ_i). *Then, if* U_{i+1} *exists,*

1. *if* U_i *is safe,* U_{i+1} *has strictly more concrete behaviors,* i.e., $\dot{U}_i \prec \dot{U}_{i+1}$,
2. *if* U_i *is unsafe,* U_{i+1} *has the same concrete behaviors,* i.e., $\dot{U}_i \preceq_{id} \dot{U}_{i+1}$ *and* $\dot{U}_{i+1} \preceq_{id} \dot{U}_i$, *and*
3. *if* U_i *is unsafe,* $\mathscr{C}_i$ *does not repeat in future,* i.e., $\forall j > i \,.\, \sigma_j(\mathscr{C}_j) \neq \sigma_i(\mathscr{C}_i)$.

In this section, we presented the high-level structure of Spacer. Many routines (InitU, ExtractInvs, Abstract, NextU, Refine, IsFeasible) are only presented by their interfaces with their implementation left open. In the next section, we complete the picture by describing the implementation used in our prototype.

5 Implementation

Let $P = \langle L, \ell^o, \ell^e, V, \tau \rangle$ be the input program. First, we transform P to $\tilde{P}$ by adding new *counter* variables for the loops of P and adding extra constraints to the transitions to count the number of iterations. Specifically, for each location ℓ we introduce a counter variable c_ℓ and a bounding variable b_ℓ. Let C and B be the sets of all counter and bounding variables, respectively, and *bound* : $C \to B$ be the bijection defined as $bound(c_\ell) = b_\ell$. We define $\tilde{P} \equiv \langle L, \ell^o, \ell^e, V \cup C \cup B, \tau \wedge \tau_B \rangle$, where $\tau_B(\ell_1, \ell_2) = \bigwedge X(\ell_1, \ell_2)$ and $X(\ell_1, \ell_2)$ is the smallest set satisfying the following conditions: (a) if $\ell_1 \to \ell_2$ is a back-edge, then $\left(0 \leq c'_{\ell_2} \ \wedge \ c'_{\ell_2} = c_{\ell_2} - 1 \ \wedge \ c_{\ell_2} \leq b_{\ell_2}\right) \in X(\ell_1, \ell_2)$, (b) else, if $\ell_1 \to \ell_2$ exits the loop headed by ℓ_k, then $(c_{\ell_k} = 0) \in X(\ell_1, \ell_2)$ and (c) otherwise, if $\ell_1 \to \ell_2$ is a transition inside the loop headed by ℓ_k, then $\left(c'_{\ell_k} = c_{\ell_k}\right) \in X(\ell_1, \ell_2)$. In practice, we use optimizations to reduce the number of variables and constraints.

Global	*Trans*	$E_{i,j} \Rightarrow \tau_\Sigma(\ell_i,\ell_j) \wedge \tau_B(\ell_i,\ell_j), \quad \ell_i,\ell_j \in L$	(1)
		$N_i \Rightarrow \bigvee_j E_{j,i}, \quad \ell_i \in L$	(2)
	Invars	$\left(\bigvee_j E_{i,j}\right) \Rightarrow \varphi, \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(3)
		$N_i \Rightarrow \varphi', \quad \ell_i \in L, \varphi \in \mathcal{I}(\ell_i)$	(4)
Local	*Lemmas*	$\bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} \left(\mathcal{A}_{\ell_i,\varphi} \Rightarrow \left(\left(\bigvee_j E_{i,j}\right) \Rightarrow \varphi\right)\right)$	(5)
		$\neg \bigwedge_{\ell_i \in L, \varphi \in \pi(\ell_i)} (\mathcal{B}_{\ell_i,\varphi} \Rightarrow (N_i \Rightarrow \varphi'))$	(6)
	Assump. Lits	$\mathcal{A}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(7)
		$\neg\mathcal{B}_{\ell,\varphi}, \quad \ell \in L, \varphi \in \pi(\ell)$	(8)
	Concrete	Σ	(9)
	Bound Vals	$b \leq bvals(b), \quad b \in B$	(10)

Fig. 6. Constraints used in our implementation of SPACER

This transformation preserves safety as shown below.

Lemma 2. *P is safe iff $\tilde{P}$ is safe,* i.e.*, if $\tilde{\mathscr{C}} = \langle \bar{\ell}, \bar{s} \rangle$ is a counterexample to $\tilde{P}$, projecting $\bar{s}$ onto V gives a counterexample to P; if $\tilde{\pi}$ is a proof of $\tilde{P}$, then $\pi = \lambda \ell \cdot \{\forall B \geq 0, C \geq 0 \,.\, \varphi \mid \varphi \in \tilde{\pi}(\ell)\}$ is a safety proof for P.*

In the rest of this section, we define our abstractions and under-approximations of $\tilde{P}$ and describe our implementation of the different routines in Fig. 5.

Abstractions. Recall that $\tau(\tilde{P}) = \tau \wedge \tau_B$. W.l.o.g., assume that τ is transformed to $\exists \Sigma \,.\, (\tau_\Sigma \wedge \bigwedge \Sigma)$ for a finite set of fresh Boolean variables Σ that only appear negatively in τ_Σ. We refer to Σ as *assumptions* following SAT terminology [14]. Dropping some assumptions from $\bigwedge \Sigma$ results in an abstract transition relation, *i.e.*, $\exists \Sigma \,.\, \left(\tau_\Sigma \wedge \bigwedge \hat{\Sigma}\right)$ is an abstraction of τ for $\hat{\Sigma} \subseteq \Sigma$, denoted $\hat{\tau}(\hat{\Sigma})$. Note that $\hat{\tau}(\hat{\Sigma}) = \tau_\Sigma[\hat{\Sigma} \leftarrow \top, \Sigma \setminus \hat{\Sigma} \leftarrow \bot]$. The only abstractions of $\tilde{P}$ we consider are the ones which abstract τ and keep τ_B unchanged. That is, every abstraction $\hat{P}$ of $\tilde{P}$ is such that $\tilde{P} \preceq_{id} \hat{P}$ with $\tau(\hat{P}) = \hat{\tau}(\hat{\Sigma}) \wedge \tau_B$ for some $\hat{\Sigma} \subseteq \Sigma$. Moreover, a subset $\hat{\Sigma}$ of Σ induces an abstraction of $\tilde{P}$, denoted $\tilde{P}(\hat{\Sigma})$.

Under-Approximations. An under-approximation is induced by a subset of assumptions $\hat{\Sigma} \subseteq \Sigma$, which identifies the abstraction $\tilde{P}(\hat{\Sigma})$, and a mapping $bvals : B \to \mathbb{N}$ from B to natural numbers, which bounds the number of iterations of every loop in $\tilde{P}$. The under-approximation, denoted $U(\hat{\Sigma}, bvals)$, satisfies $U(\hat{\Sigma}, bvals) \prec_{id} \tilde{P}(\hat{\Sigma})$, with $\tau(U(\hat{\Sigma}, bvals)) = \hat{\tau}(\hat{\Sigma}) \wedge \tau_B(bvals)$ where $\tau_B(bvals)$ is obtained from τ_B by strengthening all transitions with $\bigwedge_{b \in B} b \leq bvals(b)$.

SOLVE. We implement SOLVE (see Fig. 5) by transforming the decidable under-approximation U, after restricting by the invariants to $U_{\mathcal{I}}$, to Horn-SMT [21] (the input format of μZ) and passing the result to μZ. Note that this intentionally limits the power of μZ to solve only decidable problems. In Section 6, we compare SPACER with unrestricted μZ.

```
ExtractInvsImpl(C, {A_{ℓ,φ}}_{ℓ,φ}, {B_{ℓ,φ}}_{ℓ,φ})
begin
1   |  M := ∅, X := {A_{ℓ,φ}}_{ℓ,φ}, Y := {¬B_{ℓ,φ}}_{ℓ,φ}
2   |  T := X
3   |  while (S := Mus(C, T, Y)) ≠ ∅ do
4   |  |  M := M ∪ S, Y := Y \ M
5   |  |  X := {A_{ℓ,φ} | ¬B_{ℓ,φ} ∈ Y}
6   |  |  T := X ∪ M
7   |  return X
```

```
Mus(C, T, V)
begin
8   |  R := ∅
9   |  while Sat(C, T ∪ R) do
10  |  |  m := GetModel(C, T ∪ R)
11  |  |  R := R ∪ {v ∈ V | m(¬v)}
12  |  return R
```

Fig. 7. Our implementation of ExtractInvs of Fig. 5

We implement the routines of Spacer in Fig. 5 by maintaining a set of constraints $\mathcal{C}$ as shown in Fig. 6. Initially, $\mathcal{C}$ is *Global. Trans* encodes the transition relation of $\tilde{P}$, using fresh Boolean variables for transitions and locations ($E_{i,j}$, N_i, respectively) enforcing that a location is reachable only via one of its (incoming) edges. Choosing an abstract or concrete transition relation is done by adding a subset of Σ as additional constraints. *Invars* encodes currently known invariants. They approximate the reachable states by adding constraints for every invariant at a location in terms of current-state variables (3) and next-state variables (4). The antecedent in (3) specifies that at least one transition from ℓ_i has been taken implying that the current location is ℓ_i and the antecedent in (4) specifies that the next location is ℓ_i.

$\mathcal{C}$ is modified by each routine as needed by adding and retracting some of the *Local* constraints (see Fig. 6) as discussed below.

For a set of *assumption literals* $\mathcal{A}$, let $\textsc{Sat}(\mathcal{C}, \mathcal{A})$ be a function that checks whether $\mathcal{C} \cup \mathcal{A}$ is satisfiable, and if not, returns an *unsat core* $\hat{\mathcal{A}} \subseteq \mathcal{A}$ such that $\mathcal{C} \cup \hat{\mathcal{A}}$ is unsatisfiable.

In the rest of the section, we assume that π is a safety proof of $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$.

InitU. The initial under-approximation is $U(\Sigma, \lambda b \in B \,.\, 0)$.

ExtractInvs is implemented by ExtractInvsImpl shown in Fig. 7. It extracts a *Maximal Inductive Subset* (MIS) of the lemmas in π w.r.t. the concrete transition relation $\tau \wedge \tau_B$ of $\tilde{P}$. First, the constraints *Concrete* in Fig. 6 are added to $\mathcal{C}$, including all of Σ. Second, the constraints *Lemmas* in Fig. 6 are added to $\mathcal{C}$, where fresh Boolean variables $\mathcal{A}_{\ell,\varphi}$ and $\mathcal{B}_{\ell,\varphi}$ are used to mark every lemma φ at every location $\ell \in L$. This encodes the negation of the inductiveness condition of a safety proof (see Def. 2).

The MIS of π corresponds to the *maximal* subset $I \subseteq \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ such that $\mathcal{C} \cup I \cup \{\neg\mathcal{B}_{\ell,\varphi} \mid \mathcal{A}_{\ell,\varphi} \notin I\}$ is unsatisfiable. I is computed by ExtractInvsImpl in Fig. 7. Each iteration of ExtractInvsImpl computes a *Minimal Unsatisfiable Subset* (MUS) to identify (a minimal set of) more non-inductive lemmas (lines 3–6). M, on line 4, indicates the cumulative set of non-inductive lemmas and X, on line 5, indicates all the other lemmas. $\textsc{Mus}(\mathcal{C}, T, V)$ in Fig. 7 iteratively computes a minimal subset, R, of V such that $\mathcal{C} \cup T \cup R$ is unsatisfiable.

Abstract finds a $\hat{\Sigma}_1 \subseteq \Sigma$ such that $U_{\mathcal{I}}(\hat{\Sigma}_1, \mathit{bvals})$ is safe with proof π. The constraints *Lemmas* in Fig. 6 are added to $\mathcal{C}$ to encode the negation of the

conditions in Definition 2. Then, the constraints in *Bound Vals* in Fig. 6 are added to $\mathcal{C}$ to encode the under-approximation. This reduces the check for π to be a safety proof to that of unsatisfiability of a formula. Finally, $\textsc{Sat}(\mathcal{C}, \Sigma \cup \{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi} \cup \{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi})$ is invoked. As $\mathcal{C}$ is unsatisfiable assuming Σ and using all the lemmas (since π proves $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$), it returns an unsat core. Projecting the core onto Σ gives us $\hat{\Sigma}_1 \subseteq \Sigma$ which identifies the new abstraction and, together with *bvals*, the corresponding new under-approximation. The minimality of $\hat{\Sigma}_1$ depends on the algorithm for extracting an unsat core, which is part of the SMT engine of Z3 in our case. In practice, we use a *Minimal Unsatisfiable Subset* (MUS) algorithm to find a minimal $\hat{\Sigma}_1$. As we treat $\{\mathcal{A}_{\ell,\varphi}\}_{\ell,\varphi}$ and $\{\mathcal{B}_{\ell,\varphi}\}_{\ell,\varphi}$ as assumption literals, this also corresponds to using only the necessary lemmas during abstraction.

NextU. Given the current valuation *bvals* and the new abstraction $\hat{\Sigma}$, this routine returns $U(\hat{\Sigma}, \lambda b \in B \,.\, \mathit{bvals}(b) + 1)$.

Refine and IsFeasible. Let $U_{\mathcal{I}}(\hat{\Sigma}, \mathit{bvals})$ be unsafe with a counterexample $\mathscr{C}$. We create a new set of constraints $\mathcal{C}_{\mathscr{C}}$ corresponding to the unrolling of $\tau_\Sigma \wedge \tau_B$ along the control path of $\mathscr{C}$ and check $\textsc{Sat}(\mathcal{C}_{\mathscr{C}}, \Sigma)$. If the path is feasible in $\tilde{P}$, we find a counterexample to safety in $\tilde{P}$. Otherwise, we obtain an unsat core $\hat{\Sigma}_1 \subseteq \Sigma$ and refine the abstraction to $\hat{\Sigma} \cup \hat{\Sigma}_1$. The under-approximation is refined accordingly with the same *bvals*.

We conclude the section with a discussion of the implementation choices. NextU is implemented by incrementing all bounding variables uniformly. An alternative is to increment the bounds only for the loops whose invariants are not inductive (e.g., [2,26]). However, we leave the exploration of such strategies for future. Our use of μZ is sub-optimal since each call to Solve requires constructing a new Horn-SMT problem. This incurs an unnecessary pre-processing overhead that can be eliminated by a tighter integration with μZ. For Abstract and ExtractInvs, we use a single SMT-context with a single copy of the transition relation of the program (without unrolling it). The context is preserved across iterations of Spacer. Constraints specific to an iteration are added and retracted using the incremental solving API of Z3. This is vital for performance. For Refine and IsFeasible, we unroll the transition relation of the program along the control path of the counterexample trace returned by μZ. We experimented with an alternative implementation that instead validates each individual step of the counterexample using the same global context as Abstract. While this made each refinement step faster, it increased the number of refinements, becoming inefficient overall.

6 Experiments

We implemented Spacer in Python using Z3 v4.3.1 (with a few modifications to Z3 API[4]). The implementation and complete experimental results are available at `http://www.cs.cmu.edu/~akomurav/projects/spacer/home.html`.

[4] Our changes are being incorporated into Z3, and will be available in future versions.

Benchmarks. We evaluated SPACER on the benchmarks from the *systemc*, *product-lines*, *device-drivers-64* and *control-flow-integers* categories of SV-COMP'13. Other categories require bit-vector and heap reasoning that are not supported by SPACER. We used the front-end of UFO [3] to convert the benchmarks from C to the Horn-SMT format of μZ.

Overall, there are 1,990 benchmarks (1,591 SAFE, and 399 UNSAFE); 1,382 are decided by the UFO front-end that uses common compiler optimizations to reduce the problem. This left 608 benchmarks (231 SAFE, and 377 UNSAFE).

For the UNSAFE benchmarks, 369 cases are solved by both μZ and SPACER; in the remaining 8 benchmarks, 6 are solved by neither tool, and 2 are solved by μZ but not by SPACER. In summary, abstraction did not help for UNSAFE benchmarks and, in a few cases, it hurts significantly. Having said that, these benchmarks are easy with SPACER needing at most 3 minutes each, for the 369 cases it solves.

For the SAFE benchmarks, 176 are solved in under a minute by both tools. For them, the difference between SPACER and μZ is not significant to be meaningful. Of the remaining 55 hard benchmarks 42 are solved by either μZ, SPACER or both with a time limit of 15 minutes and 2GB of memory. The rest remain unsolved. All experiments were done on an Intel® Core™2 Quad CPU of 2.83GHz and 4GB of RAM.

Results. Table 1 shows the experimental results on the 42 solved benchmarks. The t columns under μZ and SPACER show the running times in seconds with 'TO' indicating a time-out and a 'MO' indicating a mem-out. The best times are highlighted in bold. Overall, abstraction helps for *hard* benchmarks. Furthermore, in `elev_13_22`, `elev_13_29` and `elev_13_30`, SPACER is successful even though μZ runs out of memory, showing a clear advantage of abstraction. Note that `gcnr`, under *misc*, in the table is the example from Fig. 1(a).

The B column in the table shows the final values of the loop bounding variables under the mapping *bvals*, *i.e.*, the maximum number of loop iterations (of any loop) that was necessary for the final safety proof. Surprisingly, they are very small in many of the hard instances in *systemc* and *product-lines* categories.

Columns a_f and a_m show the sizes of the final and maximal abstractions, respectively, measured in terms of the number of the original constraints used. Note that this only corresponds to the *syntactic* abstraction (see Section 4). The final abstraction done by SPACER is very aggressive. Many constraints are irrelevant with often, more than 50% of the original constraints abstracted away. Note that this is in addition to the aggressive property-independent abstraction done by the UFO front-end. Finally, the difference between a_f and a_m is insignificant in all of the benchmarks.

Another approach to ABSTRACT is to restrict abstraction to state-variables by making assignments to some next-state variables non-deterministic, as done by Vizel et al. [29] in a similar context. This was especially effective for *ssh* and *ssh-simplified* categories – see the entries marked with '*' under column t.

An alternative implementation of REFINE is to concretize the under-approximation (by refining $\hat{\Sigma}$ to Σ) whenever a spurious counterexample is found. This is analogous to Proof-Based Abstraction (PBA) [27] in hardware

Table 1. Comparison of μZ and SPACER. t and t_p are running times in seconds; B and B_p are the final values of the bounding variables; a_f and a_m are the fractions of assumption variables in the final and maximal abstractions, respectively.

Benchmark	μZ	SPACER					
	t (*sec*)	t (*sec*)	B	a_f (%)	a_m (%)	t_p (*sec*)	B_p
systemc							
pipeline	224	**120**	4	33	33	249	4
tk_ring_06	64	**48**	2	59	59	65	2
tk_ring_07	69	120	2	59	59	†**67**	2
tk_ring_08	232	**158**	2	57	57	358	2
tk_ring_09	817	**241**	2	59	59	266	2
mem_slave_1	536	**430**	3	24	34	483	2
toy	TO	822	4	32	44	†**460**	4
pc_sfifo_2	**73**	137	2	41	41	TO	–
product-lines							
elev_13_21	TO	**174**	2	7	7	TO	–
elev_13_22	MO	**336**	2	9	9	624	4
elev_13_23	TO	**309**	4	6	14	TO	–
elev_13_24	TO	**591**	4	9	9	TO	–
elev_13_29	MO	**190**	2	6	10	TO	–
elev_13_30	MO	**484**	3	11	13	TO	–
elev_13_31	TO	**349**	4	8	17	TO	–
elev_13_32	TO	**700**	4	9	9	TO	–
elev_1_21	**102**	136	11	61	61	161	11
elev_1_23	**101**	276	11	61	61	†140	11
elev_1_29	92	199	11	61	62	†**77**	11
elev_1_31	127	135	11	62	62	†**92**	11
elev_2_29	**18**	112	11	56	56	†26	11
elev_2_31	**16**	91	11	57	57	†22	11

Benchmark	μZ	SPACER					
	t (*sec*)	t (*sec*)	B	a_f (%)	a_m (%)	t_p (*sec*)	B_p
ssh							
s3_clnt_3	109	∗90	12	13	13	**73**	12
s3_srvr_1	187	**43**	9	18	18	661	25
s3_srvr_2	587	∗**207**	14	3	7	446	15
s3_srvr_8	99	**49**	13	18	18	TO	–
s3_srvr_10	83	**24**	9	17	17	412	21
s3_srvr_13	355	∗**298**	15	8	8	461	15
s3_clnt_2	**34**	∗124	13	13	13	†95	13
s3_srvr_12	**21**	∗64	13	8	8	54	13
s3_srvr_14	**37**	∗141	17	8	8	†91	17
s3_srvr_6	**98**	TO	–	–	–	†300	25
s3_srvr_11	**270**	896	15	14	18	831	13
s3_srvr_15	**309**	TO	–	–	–	TO	–
s3_srvr_16	**156**	∗263	21	8	8	†159	21
ssh-simplified							
s3_srvr_3	171	130	11	21	21	**116**	12
s3_clnt_3	**50**	∗139	12	17	22	†104	13
s3_clnt_4	**15**	∗76	12	22	22	56	13
s3_clnt_2	**138**	509	13	26	26	†145	13
s3_srvr_2	**148**	232	12	16	23	222	15
s3_srvr_6	**91**	TO	–	–	–	†272	25
s3_srvr_7	**253**	398	10	20	26	764	10
misc							
gcnr	TO	56	26	81	95	**50**	25

verification. Run-time for PBA and the corresponding final values of the bounding variables are shown in columns t_p and B_p of Table 1, respectively. While this results in more time-outs, it is significantly better in 14 cases (see the entries marked with '†' under column t_p), with 6 of them comparable to μZ and 2 (*viz.*, `toy` and `elev_1_31`) significantly better than μZ.

We conclude this section by comparing our results with UFO [3] — the winner of the 4 categories at SV-COMP'13. The competition version of UFO runs several engines in parallel, including engines based on Abstract Interpretation, Predicate Abstraction and 2BMC with interpolation. UFO outperforms SPACER and μZ in *ssh* and *product-lines* categories by an order of magnitude. They are difficult for 2BMC, but easy for Abstract Interpretation and Predicate Abstraction, respectively. Even so, note that SPACER finds really small abstractions for these categories upon termination. However, in the *systemc* category both SPACER and μZ perform better than UFO by solving hard instances (e.g., `tk_ring_08` and `tk_ring_09`) that are not solved by any tool in the competition. Moreover, SPACER is faster than μZ. Thus, while SPACER itself is not the best tool for all benchmarks, it is a valuable addition to the state-of-the-art verification engines.

7 Related Work

There is a large body of work on 2BMC approaches both in hardware and software verification. In this section, we briefly survey the most related work.

The two most prominent approaches to 2BMC combine BMC with interpolation (e.g., [2,25,26]) or with inductive generalization (e.g., [9,10,13,21]). Although our implementation of SPACER is based on inductive generalization (the engine of μZ), it can be implemented on top of an interpolation-based engine as well.

Proof-based Abstraction (PBA) was first introduced in hardware verification to leverage the power of SAT-solvers to focus on relevant facts [19,27]. Over the years, it has been combined with CEGAR [4,5], interpolation [5,24], and PDR [22]. To the best of our knowledge, SPACER is the first application of PBA to software verification.

The work of Vizel et al. [29], in hardware verification, that extends PDR with abstraction is closest to ours. However, SPACER is not tightly coupled with PDR, which makes it more general, but possibly, less efficient. Nonetheless, SPACER allows for a rich space of abstractions, whereas Vizel et al. limit themselves to state variable abstraction.

Finally, UFO [2,1] also combines abstraction with 2BMC, but in an orthogonal way. In UFO, abstraction is used to guess the depth of unrolling (plus useful invariants), BMC to detect counterexamples, and interpolation to synthesize safe inductive invariants. While UFO performs well on many competition benchmarks, combining it with SPACER will benefit on the hard ones.

8 Conclusion

In this paper, we present an algorithm, SPACER, that combines Proof-Based Abstraction (PBA) with CounterExample Guided Abstraction Refinement (CEGAR) for verifying safety properties of sequential programs. To our knowledge, this is the first application of PBA to software verification. Our abstraction technique combines localization with invariants about the program. It is interesting to explore alternatives for such a *semantic* abstraction.

While our presentation is restricted to non-recursive sequential programs, the technique can be adapted to solving the more general Horn Clause Satisfiability problem and extended to verifying recursive and concurrent programs [16].

We have implemented SPACER in Python using Z3 and its GPDR engine μZ. The current implementation is an early prototype. It is not heavily optimized and is not tightly integrated with μZ. Nonetheless, the experimental results on 4 categories of the 2nd Software Verification Competition show that SPACER improves on both μZ and the state-of-the-art.

Acknowledgment. We thank Nikolaj Bjørner for many helpful discussions and help with μZ and the anonymous reviewers for insightful comments.

References

1. Albarghouthi, A., Gurfinkel, A., Chechik, M.: Craig Interpretation. In: Miné, A., Schmidt, D. (eds.) SAS 2012. LNCS, vol. 7460, pp. 300–316. Springer, Heidelberg (2012)

2. Albarghouthi, A., Gurfinkel, A., Chechik, M.: From Under-Approximations to Over-Approximations and Back. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 157–172. Springer, Heidelberg (2012)
3. Albarghouthi, A., Gurfinkel, A., Li, Y., Chaki, S., Chechik, M.: UFO: Verification with Interpolants and Abstract Interpretation - (Competition Contribution). In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013 (ETAPS 2013). LNCS, vol. 7795, pp. 637–640. Springer, Heidelberg (2013)
4. Amla, N., McMillan, K.L.: A Hybrid of Counterexample-Based and Proof-Based Abstraction. In: Hu, A.J., Martin, A.K. (eds.) FMCAD 2004. LNCS, vol. 3312, pp. 260–274. Springer, Heidelberg (2004)
5. Amla, N., McMillan, K.L.: Combining Abstraction Refinement and SAT-Based Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 405–419. Springer, Heidelberg (2007)
6. Babić, D., Hu, A.J.: Structural Abstraction of Software Verification Conditions. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 366–378. Springer, Heidelberg (2007)
7. Ball, T., Majumdar, R., Millstein, T., Rajamani, S.K.: Automatic Predicate Abstraction of C Programs. SIGPLAN Not. 36(5), 203–213 (2001)
8. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded Model Checking. Advances in Computers 58, 117–148 (2003)
9. Bradley, A.R.: SAT-Based Model Checking without Unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011)
10. Cimatti, A., Griggio, A.: Software Model Checking via IC3. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 277–293. Springer, Heidelberg (2012)
11. Kroening, D., Weissenbacher, G.: Counterexamples with Loops for Predicate Abstraction. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 152–165. Springer, Heidelberg (2006)
12. de Moura, L., Bjørner, N.S.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
13. Eén, N., Mishchenko, A., Brayton, R.K.: Efficient Implementation of Property Directed Reachability. In: FMCAD, pp. 125–134 (2011)
14. Eén, N., Sörensson, N.: An Extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004)
15. Flanagan, C., Leino, K.R.M.: Houdini, an Annotation Assistant for ESC/Java. In: Oliveira, J.N., Zave, P. (eds.) FME 2001. LNCS, vol. 2021, pp. 500–517. Springer, Heidelberg (2001)
16. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing Software Verifiers from Proof Rules. In: PLDI, pp. 405–416 (2012)
17. Griggio, A.: A Practical Approach to Satisfiability Modulo Linear Integer Arithmetic. JSAT 8, 1–27 (2012)
18. Gulavani, B.S., Chakraborty, S., Nori, A.V., Rajamani, S.K.: Automatically Refining Abstract Interpretations. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 443–458. Springer, Heidelberg (2008)
19. Gupta, A., Ganai, M.K., Yang, Z., Ashar, P.: Iterative Abstraction using SAT-based BMC with Proof Analysis. In: ICCAD, pp. 416–423 (2003)
20. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. SIGPLAN Not. 37(1), 58–70 (2002)

21. Hoder, K., Bjørner, N.: Generalized Property Directed Reachability. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 157–171. Springer, Heidelberg (2012)
22. Ivrii, A., Matsliah, A., Mony, H., Baumgartner, J.: IC3-Guided Abstraction. In: FMCAD (2012)
23. Jhala, R., McMillan, K.L.: A Practical and Complete Approach to Predicate Refinement. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 459–473. Springer, Heidelberg (2006)
24. Li, B., Somenzi, F.: Efficient Abstraction Refinement in Interpolation-Based Unbounded Model Checking. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 227–241. Springer, Heidelberg (2006)
25. McMillan, K.L.: Interpolation and SAT-Based Model Checking. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
26. McMillan, K.L.: Lazy Abstraction with Interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
27. McMillan, K.L., Amla, N.: Automatic Abstraction without Counterexamples. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 2–17. Springer, Heidelberg (2003)
28. McMillan, K.L., Rybalchenko, A.: Solving Constrained Horn Clauses using Interpolation. Technical Report MSR-TR-2013-6, Microsoft Research (2013)
29. Vizel, Y., Grumberg, O., Shoham, S.: Lazy Abstraction and SAT-based Reachability for Hardware Model Checking. In: FMCAD (2012)

DiVinE 3.0 – An Explicit-State Model Checker for Multithreaded C & C++ Programs*

Jiří Barnat, Luboš Brim, Vojtěch Havel, Jan Havlíček, Jan Kriho, Milan Lenčo, Petr Ročkai**, Vladimír Štill, and Jiří Weiser

Faculty of Informatics, Masaryk University
Brno, Czech Republic
divine@fi.muni.cz

Abstract. We present a new release of the parallel and distributed LTL model checker DiVinE. The major improvement in this new release is an extension of the class of systems that may be verified with the model checker, while preserving the unique DiVinE feature, namely parallel and distributed-memory processing. Version 3.0 comes with support for direct model checking of (closed) multithreaded C/C++ programs, full untimed-LTL model checking of timed automata, and a general-purpose framework for interfacing with arbitrary system modelling tools.

1 Introduction

Even though explicit-state model checking is a core method of automated formal verification, there are still major roadblocks, preventing the software development industry from fully utilising explicit-state model checkers. One is the well-known state space explosion problem, which restricts the size of systems that can be efficiently handled by a model checker. Another, possibly even more serious, is the requirement to create a separate model of the system, disconnected from its source code. This adds a substantial amount of work to the process of model checking, increasing its price and making the method less feasible industrially. The problem is compounded by relative obscurity of modelling languages.

In version 3.0, DiVinE [2–5] addresses both these problems: based on a newly developed LLVM bitcode interpreter, it can directly verify closed C/C++ programs, eliminating the extra human effort directed at modelling the system. At the same time, DiVinE 3.0 offers efficient state-space reduction techniques (Partial Order Reduction, Path Compression), combined with parallel and distributed-memory processing. This makes DiVinE suitable for verification of large systems, especially when compared to more traditional, sequential model checkers.

* This work has been partially supported by the Czech Science Foundation grant No. GAP202/11/0312.

** Petr Ročkai has been partially supported by Red Hat, Inc. and is a holder of Brno PhD Talent financial aid provided by Brno City Municipality.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 863–868, 2013.

2 Engine Improvements Since DiVinE 2.5

While the primary focus of the 3.0 release was on language support, there have been important improvements in the model-checking core as well. A major addition is the optional use of hash compaction and disk-based queues, designed to work hand-in-hand to reduce memory footprint. While hash compaction introduces a small risk of missing counter-examples, and hence results obtained with hash compaction cannot guarantee correctness, it has proven to be extremely useful in tracking down bugs in large, complex systems that cannot be entirely verified at reasonable expense with available technology. As implemented in DiVinE, hash compaction can be used with both reachability analysis and LTL model checking and is compatible with distributed-memory verification. [6]

While algorithms using traditional static partitioning and per-thread hash tables provide reasonable scalability, a single shared hash-table and dynamic work partitioning can give substantially better results, as has been demonstrated by LTSmin [9]. Hence, DiVinE 3.0 provides an experimental mode of operation using a single shared hash table. While this mode is a proof of concept and is not recommended for production use in this release, future 3.x versions of DiVinE will integrate it more tightly.

3 DVE: The Native Modelling Language

The DVE language was conceived and implemented in the early phases of development of DiVinE. Since then, it became successful in its own right as a simple yet still powerful formalism for modelling asynchronous systems and protocols. Nevertheless, the original implementation has been falling out with rapid development in other parts of DiVinE. In version 3.0, we have replaced the legacy DVE interpreter with a modern, more flexible and extensible design. Gradual, backward-compatible improvements to the DVE language are expected in the 3.x line of development.

In addition to an improved interpreter, DiVinE 3.0 has added an ability to restrict LTL model checking to (weakly) fair runs. This feature is so far unique to the DVE language, although future extensions to other input languages are planned.

4 LLVM: Model Checking Multithreaded C++

The major highlight of the new version of DiVinE is the ability to directly model-check LLVM bitcode. This in turn enables programmers to use DiVinE for model checking of closed C and C++ programs, since major C and C++ compilers[1] can produce LLVM bitcode.

[1] Clang and GCC (with a plugin) can generate both optimised and unoptimised LLVM bitcode. Compilers for other languages are available as well.

Table 1. Efficiency of LLVM bitcode reductions

model, flags	state space reduction			
	none	τ	$\tau+$	all
`peterson.c, -O0`	294193	2181	596	212
`peterson.c, -O1`	33227	491	286	278
`peterson.c, -O2`	21122	443	268	260

Userspace programs normally needs to be linked to system libraries for execution; while purely computational fragments of system libraries can be directly translated into LLVM bitcode and linked into the program for verification purposes, this is not the case with "IO" facilities (including any calls into the OS kernel). For some of these, DiVinE provides substitutes – most importantly the POSIX thread API, while other may need to be provided by the user, possibly implemented in terms of a nondeterministic choice operator (`__divine_choice`) provided by DiVinE. This means that no IO is possible (but it may be substituted by nondeterminism) and this automatically makes the program closed. Hence, no other "special" treatment is required to verify programs.

Since DiVinE provides an implementation of majority of the POSIX thread APIs (`pthread.h`), it enables verification of unmodified multithreaded programs. In particular, DiVinE explores all possible thread interleavings systematically at the level of individual bitcode instructions. This allows DiVinE, for example, to virtually prove an absence of deadlock or assertion violation in a given mutlithreaded piece of code, which is impossible with standard testing techniques.

An invocation of DiVinE that performs assertion violation check for a multithreaded program, say `my_code.cpp`, is given below. First, C++ code is compiled into a LLVM bitcode file and then `divine verify` is used to execute a search for assertion violations.

```
$ divine compile --llvm [--cflags=" < flags > "] my_code.cpp
$ divine verify my_code.bc --property=assert [-d]
```

When no assertion violation is found, the same C++ code can be compiled into a native executable using the same tools and natively executed as follows.

```
$ clang [ < flags > ] -lpthread -o my_code.exe my_code.cpp
$ ./my_code.exe
```

This approach provides high assurance that the resulting binary meets the specification, since the bitcode can be verified post-optimisation. The only sources of infidelity are the native code generator (which is relatively simple compared to the optimiser) and the actual execution environment.

Without efficient state space reductions, the state space explosion stemming from the asynchronous concurrency of the very fine-grained LLVM bitcode would be prohibitive. Therefore, DiVinE comes with very efficient reduction algorithms ($\tau+$reduction and heap symmetry reduction) [10] to facilitate verification. Efficiency of the reductions is indicated in Table 1.

The high level of assurance and a low entry barrier make this approach a very attractive combination. A set of examples (implemented in C and C++) which demonstrate the existing capabilities of the LLVM interpreter is distributed with DiVinE.

5 Timed Automata

Timed automata as used in Uppaal [7, 8] became a standard modelling formalism. The new release of DiVinE comes with the ability to perform LTL model checking and deadlock detection for real-time systems designed in Uppaal. On top of Uppaal Timed Automata Parser Library[2] (UTAP) and DBM Library[3], DiVinE implements an interpreter of timed automata, based on zone abstraction, see the scheme in Figure 1.

Both imported libraries and the new interpreter are built into the `divine` binary, allowing the tool to directly accept `.xml` files as produced by Uppaal IDE. For such the real-time systems, DiVinE is capable of performing time deadlock detection. Moreover, using the automata-based approach to LTL model checking, DiVinE allows verification of properties expressed as untimed LTL formulas over values of system data and clock variables. Since this approach does not distinguish zeno and non-zeno behaviours, some counterexamples may be spurious.

For untimed LTL model checking of real-time systems it suffices to provide the tool with an `.ltl` file of the same basename as the file describing the real-time system. If such a file is present when DiVinE is executed, it is automatically loaded and DiVinE offers to perform LTL model checking in addition to reachability analysis. Examples of real-time systems and corresponding LTL properties are part of the DiVinE distribution bundle.

6 Interface to External Interpreters

In version 3.0, DiVinE officially provides support for connecting third-party modelling formalisms. To this effect, DiVinE includes a model loader written to the Common Explicit-State Model Interface specification (CESMI). The CESMI specification defines a simple interface between the model-checking core and a loadable module representing the model. Generation of model states is driven by the needs of the model checking engine.

As a binary interface, CESMI requires a set of functions to be implemented in a form of dynamic (shared) library: this library is called a CESMI module. DiVinE's CESMI loader then connects the functions implemented in the module to the model checking engine: see also Figure 1. The two functions that must be implemented by all CESMI modules provide the initial states of the state space and generate immediate successors of any given state, respectively. A detailed

[2] `http://freecode.com/projects/libutap`
[3] `http://freecode.com/projects/libudbm`

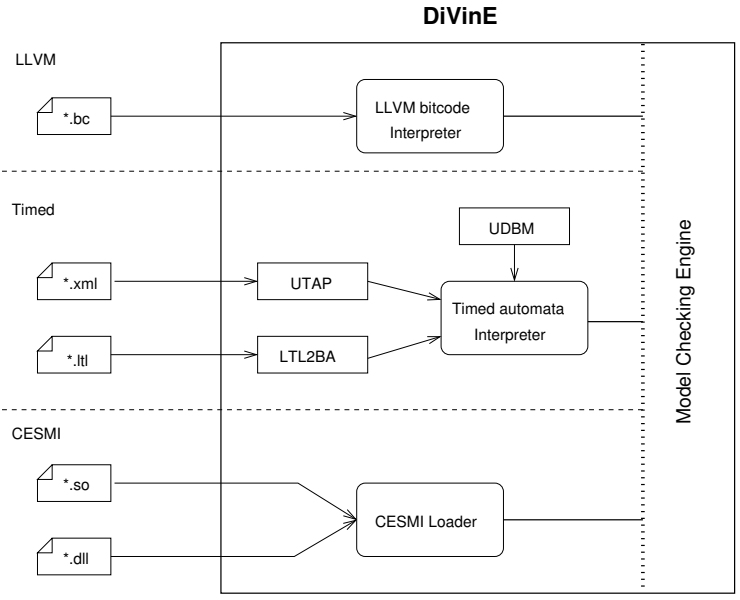


Fig. 1. Connecting DiVinE to new input languages

technical description of the interface is distributed with DiVinE. Note that the CESMI module takes different form depending on the target platform: ELF Shared Object files are supported on POSIX platforms, and Dynamically Linked Libraries (DLLs) on Win32 (Win64) platforms.

One of the advantages of using the CESMI interface in a third party project is that there is no need to implement an interpreter of the modelling language within DiVinE. In fact, new systems can be connected to DiVinE without changes to DiVinE itself, lowering the entry barrier for extending the tool.

A potential downside of the CESMI approach is that the CESMI module is responsible for presenting a Büchi automaton for the purposes of LTL model checking. While this requirement makes the CESMI specification more generic and flexible, it could present additional burden on the authors of CESMI modules. To mitigate this problem, DiVinE provides a small library of support code, automating both LTL conversion and construction of product automata. This functionality is available via the `divine compile --cesmi` sub-command and is documented in more detail in the tool manual.

The usefulness of the CESMI interface has been already demonstrated in several cases. First, we implemented a compiler of DVE (the native DiVinE modelling language) that builds CESMI modules and shows that a CESMI-based pre-compiled state generator is much faster than a run-time interpreter [5]. CESMI interface has also been successfully used in extending DiVinE to verify Murφ models [5]. More recently, the CESMI specification allowed us to build an interface between MATLAB Simulink and DiVinE, effectively creating a tool chain for verification of Simulink models [1].

7 Availability and Future Plans

DiVinE is freely available under BSD license. Stable releases as well as development snapshots and pre-releases are available for download at `divine.fi.muni.cz`.

Future development is expected to further improve scalability of the tool in parallel and distributed-memory settings. Moreover, we expect better state-space compression techniques and semi-symbolic model checking methods to again extend the applicability of DiVinE, to even larger and more complex systems. The set of C APIs offered by the LLVM interpreter will be expanded, extending the class of programs which can be verified without modification. An important future milestone is the addition of non-deterministic I/O and simulation of other system interactions.

References

1. Barnat, J., Beran, J., Brim, L., Kratochvíla, T., Ročkai, P.: Tool Chain to Support Automated Formal Verification of Avionics Simulink Designs. In: Stoelinga, M., Pinger, R. (eds.) FMICS 2012. LNCS, vol. 7437, pp. 78–92. Springer, Heidelberg (2012)
2. Barnat, J., Brim, L., Ročkai, P.: DiVinE Multi-Core – A Parallel LTL Model-Checker. In: Cha, S., Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
3. Barnat, J., Brim, L., Černá, I.: Cluster-based LTL model checking of large systems. In: de Boer, F.S., Bonsangue, M.M., Graf, S., de Roever, W.-P. (eds.) FMCO 2005. LNCS, vol. 4111, pp. 259–279. Springer, Heidelberg (2006)
4. Barnat, J., Brim, L., Černá, I., Moravec, P., Ročkai, P., Šimeček, P.: DiVinE – A Tool for Distributed Verification. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 278–281. Springer, Heidelberg (2006)
5. Barnat, J., Brim, L., Ročkai, P.: DiVinE multi-core – A parallel LTL model-checker. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 234–239. Springer, Heidelberg (2008)
6. Barnat, J., Havlíček, J., Ročkai, P.: Distributed LTL Model Checking with Hash Compaction. In: Proceedings of PASM/PDMC 2012 (to appear 2013)
7. Behrmann, G., David, A., Larsen, K.G., Möller, O., Pettersson, P., Yi, W.: Uppaal - present and future. In: Proc. of 40th IEEE Conference on Decision and Control. IEEE Computer Society Press (2001)
8. Behrmann, G., Hune, T., Vaandrager, F.W.: Distributing Timed Model Checking - How the Search Order Matters. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 216–231. Springer, Heidelberg (2000)
9. Laarman, A., van de Pol, J., Weber, M.: Multi-Core LTSmin: Marrying Modularity and Scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
10. Rockai, P., Barnat, J., Brim, L.: Improved State Space Reduction for LTL Model Checking of C & C++ Programs. In: Submitted to The 4th NASA Formal Methods Symposium (2013)

Solving Existentially Quantified Horn Clauses

Tewodros A. Beyene[1], Corneliu Popeea[1], and Andrey Rybalchenko[1,2]

[1] Technische Universität München
[2] Microsoft Research Cambridge

Abstract. Temporal verification of universal (i.e., valid for all computation paths) properties of various kinds of programs, e.g., procedural, multi-threaded, or functional, can be reduced to finding solutions for equations in form of universally quantified Horn clauses extended with well-foundedness conditions. Dealing with existential properties (e.g., whether there exists a particular computation path), however, requires solving forall-exists quantified Horn clauses, where the conclusion part of some clauses contains existentially quantified variables. For example, a deductive approach to CTL verification reduces to solving such clauses. In this paper we present a method for solving forall-exists quantified Horn clauses extended with well-foundedness conditions. Our method is based on a counterexample-guided abstraction refinement scheme to discover witnesses for existentially quantified variables. We also present an application of our solving method to automation of CTL verification of software, as well as its experimental evaluation.

1 Introduction

Temporal verification of universal, i.e., valid for all computation paths, properties of various kinds of programs is a success story. Various techniques, e.g., abstract domains [13], predicate abstraction [18,22], or interpolation [26], provide a basis for efficient tools for the verification of such properties, e.g., Astree [5], Blast [22], CPAChecker [3], SatAbs [9], Slam [2], Terminator [12], or UFO [1]. To a large extent, the success of checkers of universal properties is determined by tremendous advances in the state-of-the-art in decision procedures for (universal) validity checking, i.e., advent of tools like MathSAT [6] or Z3 [15].

In contrast, advances in dealing with existential properties of programs, e.g., proving whether there exists a particular computation path, are still not on par with the maturity of verifiers for universal properties. Nevertheless, important first steps were made in proving existence of infinite program computations, see e.g. [16, 20, 29], even in proving existential (as well as universal) CTL properties [11]. Moreover, bounded model checking tools like CBMC [8] or Klee [7] can be very effective in proving existential reachability properties. All these initial achievements inspire further, much needed research on the topic.

In this paper, we present a method that can serve as a further building block for the verification of temporal existential (and universal) properties of programs. Our method solves forall-exists quantified Horn clauses extended with

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 869–882, 2013.

well-foundedness conditions. (The conclusion part of such clauses may contain existentially quantified variables.) The main motivation for the development of our method stems from an observation that verification conditions for existential temporal properties, e.g., generated by a deductive proof system for CTL [25], can be expressed by clauses in such form.

Our method, called E-HSF, applies a counterexample-guided refinement scheme to discover witnesses for existentially quantified variables. The refinement loop collects a global constraint that declaratively determines which witnesses can be chosen. The chosen witnesses are used to replace existential quantification, and then the resulting universally quantified clauses are passed to a solver for such clauses. At this step, we can benefit from emergent tools in the area of solving Horn clauses over decidable theories, e.g., HSF [19], μZ [23], or Duality [27]. Such a solver either finds a solution, i.e., a model for uninterpreted relations constrained by the clauses, or returns a counterexample, which is a resolution tree (or DAG) representing a contradiction. E-HSF turns the counterexample into an additional constraint on the set of witness candidates, and continues with the next iteration of the refinement loop. Notably, our refinement loop conjoins constraints that are obtained for all discovered counterexamples. This way E-HSF guarantees that previously handled counterexamples are not rediscovered and that a wrong choice of witnesses can be mended.

We applied our implementation of E-HSF to forall-exists quantified Horn clauses with well-foundedness conditions that we obtained by from a deductive proof system for CTL [25]. The experimental evaluation on benchmarks from [11] demonstrates the feasibility of our method.

2 Preliminaries

In this section we introduce preliminary definitions.

Constraints. Let $\mathcal{T}$ be a first-order theory in a given signature and $\models_{\mathcal{T}}$ be the entailment relation for $\mathcal{T}$. We write $v, v_0, v_1, \ldots$ and w to denote non-empty tuples of variables. We refer to a formula $c(v)$ over variables v from $\mathcal{T}$ as a constraint. Let *false* and *true* be an unsatisfiable and a valid constraint, respectively.

For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \leq 2$, $y \leq 1 \wedge x - y \leq 0$, and $f(x) + g(x, y) \leq 3 \vee z \leq 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities and uninterpreted functions, where f and g are uninterpreted function symbols. $y \leq 1 \wedge x - y \leq 0 \models_{\mathcal{T}} x \leq 2$ is an example of a valid entailment.

A binary relation is well-founded if it does not admit any infinite chains. A relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations [31], i.e., if there exist well-founded $\varphi_1(v, v'), \ldots, \varphi_n(v, v')$ such that $\varphi(v, v') \models_{\mathcal{T}} \varphi_1(v, v') \vee \cdots \vee \varphi_n(v, v')$. For example, the relation $x \geq 0 \wedge x' \leq x - 1$ is well-founded, while the relation $(x \geq 0 \wedge x' \leq x - 1) \vee (y \leq 0 \wedge y' \geq y + 1)$ is disjunctively well-founded.

Queries and dwf-Predicates. We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. Furthermore, we introduce an interpreted predicate symbol *dwf* of arity one (*dwf* stands for disjunctive well-foundedness). Given a query $q(v, v')$ over tuples of variables with equal length, we refer to $dwf(q)$ as a *dwf*-predicate. For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries, and $dwf(s)$ is a *dwf*-predicate.

Forall-Exists Horn-Like Clauses. Let $h(v)$ range over queries over v, constraints over v, and existentially quantified conjunctions of queries and constraints with free variables in v. We define a forall-exists Horn-like clause to be either an implication $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$ or a unit clause $dwf(q)$. The left-hand side of the implication is called the body, written as $body(v)$, and the right-hand side is called the head.

We give as example a set of forall-exists Horn-like clauses below:

$$x \geq 0 \rightarrow \exists y : x \geq y \wedge rank(x, y), \qquad rank(x, y) \rightarrow ti(x, y),$$
$$ti(x, y) \wedge rank(y, z) \rightarrow ti(x, z), \qquad dwf(ti).$$

These clauses represent an assertion over the interpretation of predicate symbols *rank* and *ti*.

Semantics of Forall-Exists Horn-Like Clauses. A set of clauses can be seen as an assertion over the queries that occur in the clauses.

We consider a function *ClauseSol* that maps each query $q(v)$ occurring in a given set of clauses into a constraint over v. Such a function is called a solution if the following two conditions hold. First, for each clause of the form $body(v) \rightarrow h(v)$ from the given set we require that replacing each query by the corresponding constraint assigned by *ClauseSol* results in a valid entailment. That is, we require $body(v)ClauseSol \models_{\mathcal{T}} h(v)ClauseSol$, where the juxtaposition represents application of substitution. Second, for each clause of the form $dwf(q)$ we require that the constraint assigned by *ClauseSol* to q represents a disjunctively well-founded relation. Let $\models_{\mathcal{Q}}$ be the corresponding satisfaction relation, i.e., $ClauseSol \models_{\mathcal{Q}} Clauses$ if *ClauseSol* is a solution for the given set of clauses.

For example, the previously presented set of clauses, say *Clauses*, has a solution *ClauseSol* such that $ClauseSol(rank(x, y)) = ClauseSol(ti(x, y)) = (x \geq 0 \wedge y \geq x - 1)$. To check $ClauseSol \models_{\mathcal{Q}} Clauses$ we consider the validity of the following implications:

$$x \geq 0 \rightarrow \exists y : x \geq y \wedge x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \rightarrow x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \wedge y \geq 0 \wedge z \leq y - 1 \rightarrow x \geq 0 \wedge z \leq x - 1.$$

and the fact that $ClauseSol(ti(x, y)) = (x \geq 0 \wedge y \leq x - 1)$ is a (disjunctively) well-founded relation.

Solving Horn-like Clauses without Existential Quantification. We assume an algorithm HSF for solving Horn-like clauses whose heads do not contain any existential quantification. This algorithm computes a solution *ClauseSol* when it exists. There already exist such algorithms as well as their efficient implementations that are based on predicate abstraction and interpolation [19], as well as interpolation based approximation [27].

3 Preliminaries

In this section we introduce preliminary definitions.

Constraints. Let $\mathcal{T}$ be a first-order theory in a given signature and $\models_{\mathcal{T}}$ be the entailment relation for $\mathcal{T}$. We write $v, v_0, v_1, \ldots$ and w to denote non-empty tuples of variables. We refer to a formula $c(v)$ over variables v from $\mathcal{T}$ as a constraint. Let *false* and *true* be an unsatisfiable and a valid constraint, respectively.

For example, let x, y, and z be variables. Then, $v = (x, y)$ and $w = (y, z)$ are tuples of variables. $x \leq 2$, $y \leq 1 \wedge x - y \leq 0$, and $f(x) + g(x, y) \leq 3 \vee z \leq 0$ are example constraints in the theory $\mathcal{T}$ of linear inequalities and uninterpreted functions, where f and g are uninterpreted function symbols. $y \leq 1 \wedge x - y \leq 0 \models_{\mathcal{T}} x \leq 2$ is an example of a valid entailment.

A binary relation is well-founded if it does not admit any infinite chains. A relation $\varphi(v, v')$ is disjunctively well-founded if it is included in a finite union of well-founded relations [31], i.e., if there exist well-founded $\varphi_1(v, v'), \ldots, \varphi_n(v, v')$ such that $\varphi(v, v') \models_{\mathcal{T}} \varphi_1(v, v') \vee \cdots \vee \varphi_n(v, v')$. For example, the relation $x \geq 0 \wedge x' \leq x - 1$ is well-founded, while the relation $(x \geq 0 \wedge x' \leq x - 1) \vee (y \leq 0 \wedge y' \geq y + 1)$ is disjunctively well-founded.

Queries and dwf-predicates. We assume a set of uninterpreted predicate symbols $\mathcal{Q}$ that we refer to as query symbols. The arity of a query symbol is encoded in its name. We write q to denote a query symbol. Given q of a non-zero arity n and a tuple of variables v of length n, we define $q(v)$ to be a query. Furthermore, we introduce an interpreted predicate symbol *dwf* of arity one (*dwf* stands for disjunctive well-foundedness). Given a query $q(v, v')$ over tuples of variables with equal length, we refer to $dwf(q)$ as a *dwf*-predicate. For example, let $\mathcal{Q} = \{r, s\}$ be query symbols of arity one and two, respectively. Then, $r(x)$ and $s(x, y)$ are queries, and $dwf(s)$ is a *dwf*-predicate.

Forall-exists Horn-like Clauses. Let $h(v)$ range over queries over v, constraints over v, and existentially quantified conjunctions of queries and constraints with free variables in v. We define a forall-exists Horn-like clause to be either an implication $c(v_0) \wedge q_1(v_1) \wedge \cdots \wedge q_n(v_n) \rightarrow h(v)$ or a unit clause $dwf(q)$. The left-hand side of the implication is called the body, written as $body(v)$, and the right-hand side is called the head.

We give as example a set of forall-exists Horn-like clauses below:

$$x \geq 0 \rightarrow \exists y : x \geq y \wedge rank(x, y), \qquad rank(x, y) \rightarrow ti(x, y),$$
$$ti(x, y) \wedge rank(y, z) \rightarrow ti(x, z), \qquad dwf(ti).$$

These clauses represent an assertion over the interpretation of predicate symbols *rank* and *ti*.

Semantics of Forall-exists Horn-like Clauses. A set of clauses can be seen as an assertion over the queries that occur in the clauses.

We consider a function *ClauseSol* that maps each query $q(v)$ occurring in a given set of clauses into a constraint over v. Such a function is called a solution if the following two conditions hold. First, for each clause of the form $body(v) \rightarrow h(v)$ from the given set we require that replacing each query by the corresponding constraint assigned by *ClauseSol* results in a valid entailment. That is, we require $body(v)ClauseSol \models_{\mathcal{T}} h(v)ClauseSol$, where the juxtaposition represents application of substitution. Second, for each clause of the form $dwf(q)$ we require that the constraint assigned by *ClauseSol* to q represents a disjunctively well-founded relation. Let $\models_{\mathcal{Q}}$ be the corresponding satisfaction relation, i.e., $ClauseSol \models_{\mathcal{Q}} Clauses$ if *ClauseSol* is a solution for the given set of clauses.

For example, the previously presented set of clauses, say *Clauses*, has a solution *ClauseSol* such that $ClauseSol(rank(x, y)) = ClauseSol(ti(x, y)) = (x \geq 0 \wedge y \geq x - 1)$. To check $ClauseSol \models_{\mathcal{Q}} Clauses$ we consider the validity of the following implications:

$$x \geq 0 \rightarrow \exists y : x \geq y \wedge x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \rightarrow x \geq 0 \wedge y \leq x - 1,$$
$$x \geq 0 \wedge y \leq x - 1 \wedge y \geq 0 \wedge z \leq y - 1 \rightarrow x \geq 0 \wedge z \leq x - 1.$$

and the fact that $ClauseSol(ti(x, y)) = (x \geq 0 \wedge y \leq x - 1)$ is a (disjunctively) well-founded relation.

Solving Horn-like Clauses without Existential Quantification. We assume an algorithm HSF for solving Horn-like clauses whose heads do not contain any existential quantification. This algorithm computes a solution *ClauseSol* when it exists. There already exist such algorithms as well as their efficient implementations that are based on predicate abstraction and interpolation [19], as well as interpolation based approximation [27].

4 Example of Applying E-HSF

We consider the following set *Clauses* that encodes a check whether a program with the variables $v = (x, y)$, an initial condition $init(v) = (y \geq 1)$ and a transition relation $next(v, v') = (x' = x + y)$ satisfies a CTL property $EF\ dst(v)$, where $dst(v) = (x \geq 0)$.

$$init(v) \rightarrow inv(v), \quad inv(v) \wedge \neg dst(v) \rightarrow \exists v' : next(v, v') \wedge inv(v') \wedge rank(v, v'),$$
$$rank(v, v') \rightarrow ti(v, v'), \quad ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v''), \quad dwf(ti).$$

Here, $inv(v)$, $rank(v, v')$, and $ti(v, v')$ are unknown predicates that we need to solve for. The predicate $inv(v)$ corresponds to states reachable during program execution, while the second row of clauses ensures that $rank(v, v')$ is a well-founded relation [31].

We start the execution of E-HSF from Figure **??** by applying SKOLEMIZE to eliminate the existential quantification. As a result, the clause that contains existential quantification is replaced by the following four clauses that contain an application of a Skolem relation $rel(v, v')$ introduced by SKOLEMIZE as well as an introduction of a lower bound on the guard $grd(v)$ of the Skolem relation:

$$\begin{aligned}
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow next(v, v'),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow inv(v'),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow rank(v, v'),\\
&inv(v) \wedge \neg dst(v) \rightarrow grd(v).
\end{aligned}$$

Furthermore, this introduction is recorded as $Rels = \{rel\}$ and $Grds = \{grd\}$. Note that we replaced a conjunction in the head of a clause by a conjunction of corresponding clauses.

First Candidate for Skolem Relation. Next, we proceed with the execution of E-HSF. We initialise *Constraint* with the assertion *true*. Then, we generate a set of Horn clauses *Defs* that provides initial candidates for the Skolem relation and its guard as follows: $Defs = \{true \rightarrow rel(v, v'),\ grd(v) \rightarrow true\}$. Now, we apply the solving algorithm HSF for quantifier free Horn clauses on the set of clauses that contains the result of Skolemization and the initial candidates in *Defs*, i.e., we give to HSF the following clauses:

$$\begin{aligned}
&init(v) \rightarrow inv(v), && rank(v, v') \rightarrow ti(v, v'),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow next(v, v'), && ti(v, v') \wedge rank(v', v'') \rightarrow ti(v, v''),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow inv(v'), && dwf(ti),\\
&inv(v) \wedge \neg dst(v) \wedge rel(v, v') \rightarrow rank(v, v'), && true \rightarrow rel(v, v'),\\
&inv(v) \wedge \neg dst(v) \rightarrow grd(v), && grd(v) \rightarrow true.
\end{aligned}$$

HSF returns an error derivation that witnesses a violation of the given set of clauses. This derivation represents an unfolding of clauses in $Skolemized \cup Defs$ that yields a relation for $ti(v, v')$ that is not disjunctively well-founded. To represent the unfolding, HSF uses a form of static single assignment (SSA) that is applied to predicate symbols, where each unfolding step introduces a fresh predicate symbol that is recorded by the function SYM. We obtain the clauses *Cex* consisting of

$$init(v) \rightarrow q_1(v),\quad q_1(v) \wedge \neg dst(v) \wedge q_2(v, v') \rightarrow next(v, v'),\quad true \rightarrow q_2(v, v')$$

together with the following bookkeeping of the SSA renaming: $\text{SYM}(q_1) = inv$ and $\text{SYM}(q_2) = rel$. From *Cex* we extract the clause *CexDefs* that provides the candidate for the Skolem relation. We obtain $CexDefs = \{true \rightarrow q_2(v, v')\}$, since $\text{SYM}(q_2) = rel$ and hence $\text{SYM}(q_2) \in Rels$.

We analyse the counterexample clauses by applying resolution on $Cex \setminus CexDefs$. The corresponding resolution tree is shown below (literals selected for resolution are boxed):

$$\frac{init(v) \rightarrow \boxed{q_1(v)} \qquad \boxed{q_1(v)} \wedge \neg dst(v) \wedge q_2(v,v') \rightarrow next(v,v')}{init(v) \wedge \neg dst(v) \wedge q_2(v,v') \rightarrow next(v,v')}$$

Note that $q_2(v,v')$ was not eliminated, since the clause $true \rightarrow q_2(v,v')$ was not given to RESOLVE as input. The result of applying RESOLVE is the clause $init(v) \wedge \neg dst(v) \wedge q_2(v,v') \rightarrow next(v,v')$. We assign the conjunction $init(v) \wedge \neg dst(v)$ to *body* and $next(v,v')$ to *head*, respectively.

Now we iterate i through the singleton set $\{1\}$, which is determined by the fact that the above clause contains only one unknown predicate on the left-hand side. We apply RELT on $\text{SYM}(q_2)$ and set the free variables in the result to (v,v'). This yields a template $v' = Tv + t$ for the Skolem relation $rel(v,v')$. Here, T is a matrix of unknown coefficients $\left(\begin{smallmatrix} t_{xx} & t_{xy} \\ t_{yx} & t_{yy} \end{smallmatrix}\right)$, and t is a vector of unknown free coefficient (t_x, t_y). In other words, our template represents a conjunction of two equality predicates $x' = t_{xx}x + t_{xy}y + t_x$ and $y' = t_{yx}x + t_{yy}y + t_y$. We conjoin this template with *body* and obtain $body = (v' = Tv + t \wedge init(v) \wedge \neg dst(v))$. Since *head* is not required to be disjunctively well-founded, E-HSF proceeds with the generation of constraints over template parameters.

We apply ENCODEVALIDITY on the following implication:

$$x' = t_{xx}x + t_{xy}y + t_x \wedge y' = t_{yx}x + t_{yy}y + t_y \wedge y \geq 1 \wedge \neg x \geq 0 \rightarrow x' = x + y \ .$$

This implication is valid if the following constraint returned by ENCODEVALIDITY is satisfiable.

$$\exists \overbrace{\lambda_1, \lambda_2, \lambda_3, \lambda_4}^{\lambda}, \overbrace{\mu_1, \mu_2, \mu_3, \mu_4}^{\mu} : \lambda_3 \geq 0 \wedge \lambda_4 \geq 0 \wedge \mu_3 \geq 0 \wedge \mu_4 \geq 0 \wedge$$
$$\begin{pmatrix} \lambda \\ \mu \end{pmatrix} \begin{pmatrix} t_{xx} & t_{xy} & -1 & 0 \\ t_{yx} & t_{yy} & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} -1 & -1 & 1 & 0 \\ 1 & 1 & -1 & 0 \end{pmatrix} \wedge \begin{pmatrix} \lambda \\ \mu \end{pmatrix} \begin{pmatrix} -t_x \\ -t_y \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$$

This constraint requires that the right-hand side on the implication is obtained as a linear combination of the (in)equalities on the left-hand side of the implication. We conjoin the above constraint with *Constraint*.

We apply an SMT solver to compute a satisfying valuation of template parameters occurring in *Constraint* and obtain:

t_{xx}	t_{xy}	t_x	t_{yx}	t_{yy}	t_y
1	1	0	0	0	10

By applying *CexSol* on the template $v' = Tv + t$, which is the result of $\text{RELT}(rel)(v,v')$, we obtain the conjunction $x' = x + y \wedge y' = 10$. In this example,

we assume that the template $\textsc{Grd}\textsc{T}(grd)(v)$ is equal to *true*. Hence, we modify the clauses that record the current candidate for $rel(v, v')$ and $grd(v)$ as follows:

$$Defs = \{x' = x + y \wedge y' = 10 \rightarrow rel(v, v'),\ \ grd(v) \rightarrow true\}$$

Now we proceed with the next iteration of the main loop in E-HSF.

Second Candidate for Skolem Relation. The second iteration in E-HSF uses *Defs* and *Constraint* as determined during the first iteration. We apply HSF on $Skolemized \cup Defs$ and obtain an error derivation *Cex* consisting of the clauses

$$init(v) \wedge q_1(v), \quad q_1(v) \wedge \neg dst(v) \wedge q_2(v, v') \rightarrow q_3(v, v'),$$
$$x' = x + y \wedge y' = 10 \rightarrow q_2(v, v'), \quad q_3(v, v') \rightarrow q_4(v, v'),$$

together with the function Sym such that $\textsc{Sym}(q_1) = inv$, $\textsc{Sym}(q_2) = rel$, $\textsc{Sym}(q_3) = rank$, and $\textsc{Sym}(q_4) = ti$. From *Cex* we extract $CexDefs = \{x' = x + y \wedge y' = 10 \rightarrow q_2(v, v')\}$ since $\textsc{Sym}(q_2) \in Rels$. We apply Resolve on $Cex \setminus CexDefs$ and obtain:

$$init(v) \wedge \neg dst(v) \wedge q_2(v, v') \rightarrow q_4(v, v')\ .$$

As seen at the first iteration, we have $\textsc{Rel}\textsc{T}(rel)(v, v') = (v' = Tv + t)$. Hence we have $body = (init(v) \wedge \neg dst(v) \wedge v' = Tv + t)$.

Since $\textsc{Sym}(q_4) = ti$ and $dwf(ti) \in Skolemized$, the error derivation witnesses a violation of disjunctive well-foundedness. Hence, by applying BoundT and DecreaseT we construct templates $bound(v)$ and $decrease(v, v')$ corresponding to a bound and decrease condition over the program variables, respectively.

$$bound(v) = (r_x x + r_y y \geq r_0)\ ,$$
$$decrease(v, v') = (r_x x' + r_y y' \leq r_x x + r_y y - 1)\ .$$

Finally, we set *head* to the conjunction $r_x x + r_y y \geq r_0 \wedge r_x x' + r_y y' \leq r_x x + r_y y - 1$.

By EncodeValidity on the implication $body \rightarrow head$ we obtain the constraint

$$\exists \overbrace{\lambda_1, \lambda_2, \lambda_3, \lambda_4}^{\lambda}, \overbrace{\mu_1, \mu_2, \mu_3, \mu_4}^{\mu} : \lambda_3 \geq 0 \wedge \lambda_4 \geq 0 \wedge \mu_3 \geq 0 \wedge \mu_4 \geq 0 \wedge$$
$$\begin{pmatrix}\lambda \\ \mu\end{pmatrix} \begin{pmatrix} t_{xx} & t_{xy} & -1 & 0 \\ t_{yx} & t_{yy} & 0 & -1 \\ 0 & -1 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} -r_x & -r_y & 0 & 0 \\ -r_x & -r_y & r_x & r_y \end{pmatrix} \wedge \begin{pmatrix}\lambda \\ \mu\end{pmatrix} \begin{pmatrix} -t_x \\ -t_y \\ -1 \\ -1 \end{pmatrix} = \begin{pmatrix} -r_0 \\ -1 \end{pmatrix}.$$

We add the above constraint as an additional conjunct to *Constraint*. That is, *Constraint* is strengthened during each iteration.

We apply the SMT solver to compute a valuation template parameters that satisfies *Constraint*. We obtain the following solution *CexSol*:

t_{xx}	t_{xy}	t_x	t_{yx}	t_{yy}	t_y
1	0	1	0	0	1

The corresponding values of r and r_0 are $(-1, 0)$ and -1, which lead to the bound $-x \geq 1$ and the decrease relation $-x' \leq -x - 1$. By applying *CexSol* on the template $v' = Tv + t$ we obtain the conjunction $x' = x + 1 \wedge y' = 1$. Note that the solution for $rel(v, v')$ obtained at this iteration is not compatible with the solution obtained at the first iteration, i.e., the intersection of the respective Skolem relations is empty. Finally, we modify *Defs* according to *CexSol* and obtain:

$$Defs = \{x' = x + 1 \wedge y' = 1 \rightarrow rel(v, v'),\ grd \rightarrow true\}$$

Now we proceed with the next iteration of the main loop in E-HSF. At this iteration the application of HSF returns a solution *ClauseSol* such that

$$\begin{aligned} ClauseSol(inv(v)) &= (y \geq 1)\ , \\ ClauseSol(rel(v)) &= (x' = x + 1 \wedge y' = 1)\ , \\ ClauseSol(rank(v, v')) &= (x \leq -1 \wedge x' \geq x + 1)\ , \\ ClauseSol(ti(v, v')) &= (x \leq -1 \wedge x' \geq x + 1)\ . \end{aligned}$$

Thus, the algorithm E-HSF finds a solution to the original set of forall-exists Horn clauses (and hence proves the program satisfies the CTL property).

5 Verifying CTL Properties Using E-HSF

In this section we show how E-HSF can be used for automatically proving CTL properties of programs. We utilize a standard reduction step from CTL properties to existentially quantified Horn-like clauses with well-foundedness conditions, see e.g. [25]. Here, due to space constraints, we only illustrate the reduction, using examples and refer to [25] for details of the CTL proof system.

We consider a program over variables v, with an initial condition given by an assertion $init(v)$, and a transition relation $next(v, v')$. Given a CTL property, we generate Horn-like clauses such that the property is satisfied if and only if the set of clauses is satisfiable.

The generation proceeds in two steps. The first step decomposes the property into sub-properties by following the nesting structure of the path quantifiers that occur in the property. As a result we obtain a set of simple CTL formulas that contain only one path quantifier. Each property is accompanied by a predicate that represents a set of program states that needs to be discovered.

As an example, we present the decomposition of $(init(v), next(v, v')) \models_{CTL} AG(EF(dst(v)))$, where $dst(v)$ is a first-order assertion over v. Since $EF(dst(v))$ is a sub-formula with a path quantifier as the outmost symbol, we introduce a fresh predicate $p(v)$ that is used to replace $EF(dst(v))$. Furthermore, we require that every computation that starts in a state described by $p(v)$ satisfies $EF(dst(v))$. Since the resulting CTL formulas do not have any nested path quantifiers we stop the decomposition process. The original verification question is equivalent to the existence of $p(v)$ such that $(init(v), next(v, v')) \models_{CTL} AG(p(v))$ and $(p(v), next(v, v')) \models_{CTL} EF(dst(v))$.

At the second step we consider each of the verification sub-questions obtained by decomposing the property and generate Horn-like clauses that constrain auxiliary sets and relations over program states. For $(init(v), next(v, v')) \models_{CTL} AG(p(v))$ we obtain the following clauses over an auxiliary predicate $inv_1(v)$:

$$init(v) \rightarrow inv_1(v), \quad inv_1(v) \wedge next(v, v') \rightarrow inv_1(v'), \quad inv_1(v) \rightarrow p(v).$$

Due to the existential path quantifier in $(p(v), next(v, v')) \models_{CTL} EF(dst(v))$ we obtain clauses that contain existential quantification. We deal with the eventuality by imposing a well-foundedness condition. The resulting clauses over auxiliary $inv_2(v)$, $rank(v, v')$, and $ti(v, v')$ are below (note that $dst(v)$ is a constraint, and hence can occur under negation).

$$p(v) \rightarrow inv_2(v), \quad inv_2(v) \wedge \neg dst(v) \rightarrow \exists v' : next(v, v') \wedge rank(v, v'),$$
$$rank(v, v') \rightarrow ti(v, v'), \quad ti(v, v') \wedge rank(v, v') \rightarrow ti(v, v''), \quad dwf(ti).$$

Finally, the above clauses have a solution for $inv_1(v)$, $p(v)$, $inv_2(v)$, $rank(v, v')$, and $ti(v, v')$ if and only if $(init(v), next(v, v')) \models_{CTL} AG(EF(dst(v)))$. Then, we apply E-HSF as a solver.

6 Experiments

In this section we present our implementation of E-HSF and its experimental evaluation on proving universal and existential CTL properties of programs.

Our implementation relies on HSF [19] to solve universally-quantified Horn clauses over linear inequalities (see line 4 in Figure **??**) and on the Z3 solver [15] at line 19 in Figure **??** to solve (possibly non-linear) constraints. The input to our tool is a transition system described using Prolog facts $init(v)$ and $next(v, v')$, as well as forall-exists Horn clauses corresponding to the CTL property to be proved or disproved.

We run E-HSF on the examples from industrial code from [11, Figure 7]: `OS frag.1`, `OS frag.2`, `OS frag.3`, `OS frag.4`, `OS frag.5`, `PgSQL arch` and `S/W Updates`. For each pair of a program and CTL property ϕ , we generated two verification tasks: prove ϕ and prove $\neg\phi$. The existence of a proof for a property ϕ implies that $\neg\phi$ is violated by the same program. (Similarly, a proof for $\neg\phi$ implies that ϕ is violated by the same program.)

GrdT and RelT are provided by the user and need to satisfy Equation 1. Currently, this condition is not checked by the implementation, but could be done for linear templates using quantifier elimination techniques. For our examples, linear templates are sufficiently expressive. We use $\text{RelT}(next)(v, v') = (next(v, v') \wedge w' = Tv + t \wedge Gv \leq g)$ and $\text{GrdT}(next)(v, v') = (Gv \leq g \wedge \exists v' : next(v, v'))$, where w is a subset of v that is left unconstrained by $next(v, v')$. Such w' are explicitly marked in the original benchmark programs using names `rho1`, `rho2`, ... For direct comparison with the results from [11], we used template functions corresponding to the `rho`-variables. The quantifier

Table 1. Evaluation of E-HSF on industrial benchmarks from [11]. Each "Name" column gives the corresponding program number in [11, Figure 7]. For P12, E-HSF returns different results compared to [11]. For P26, P27 and P28 both properties ϕ and $\neg\phi$ are satisfied only for some initial states. (Neither ϕ nor $\neg\phi$ hold for these programs.)

Program	Property ϕ	$\models_{CTL}\ \phi$			$\models_{CTL}\ \neg\phi$		
		Result	Time	Name	Result	Time	Name
P1	$AG(a = 1 \rightarrow AF(r = 1))$	✓	1.2s	1	×	2.7s	29
P2	$EF(a = 1 \wedge EG(r \neq 5))$	✓	0.6s	30	×	5.2s	2
P3	$AG(a = 1 \rightarrow EF(r = 1))$	✓	4.8s	3	×	0.1s	31
P4	$EF(a = 1 \wedge AG(r \neq 1))$	✓	0.6s	32	×	0.4s	4
P5	$AG(s = 1 \rightarrow AF(u = 1))$	✓	6.1s	5	×	0.2s	33
P6	$EF(s = 1 \wedge EG(u \neq 1))$	✓	1.4s	34	×	3.6s	6
P7	$AG(s = 1 \rightarrow EF(u = 1))$	✓	12.9s	7	×	0.2s	35
P8	$EF(s = 1 \wedge AG(u \neq 1))$	✓	44.7s	36	×	3.8s	8
P9	$AG(a = 1 \rightarrow AF(r = 1))$	✓	51.3s	9	×	120.0s	37
P10	$EF(a = 1 \wedge EG(r \neq 1))$	✓	132.0s	38	×	45.9s	10
P11	$AG(a = 1 \rightarrow EF(r = 1))$	✓	67.6s	11	×	3.9s	39
P12	$EF(a = 1 \wedge AG(r \neq 1))$	✓	67.9s	12	×	3.8s	40
P13	$AF(io = 1) \vee AF(ret = 1)$	✓	37m54s	13	T/O	-	41
P14	$EG(io \neq 1) \wedge EG(ret \neq 1)$	T/O	-	42	×	136.6s	14
P15	$EF(io = 1) \wedge EF(ret = 1)$	T/O	-	15	×	1.4s	43
P16	$AG(io \neq 1) \vee AG(ret \neq 1)$	✓	0.1s	44	×	874.5s	16
P17	$AG(AF(w \geq 1))$	✓	3.0s	17	×	0.1s	45
P18	$EF(EG(w < 1)$	✓	0.5s	46	×	3.5s	18
P19	$AG(EF(w \geq 1))$	✓	3.3s	19	×	0.1s	47
P20	$EF(AG(w < 1)$	✓	0.7s	48	×	0.1s	20
P21	$AG(AF(w = 1)$	✓	2.8s	21	×	0.1s	49
P22	$EF(EG(w \neq 1)$	✓	2.2s	50	×	5.0s	22
P23	$AG(EF(w = 1)$	✓	4.5s	23	×	0.1s	51
P24	$EF(AG(w \neq 1)$	✓	3.4s	52	×	0.7s	24
P25	$c > 5 \rightarrow AF(r > 5)$	✓	3.2s	25	×	0.1s	53
P26	$c > 5 \wedge EG(r \leq 5)$	×	0.1s	54	×	1.3s	26
P27	$c > 5 \rightarrow EF(r > 5)$	×	0.2s	27	×	0.1s	55
P28	$c > 5 \wedge AG(r \leq 5)$	×	0.1s	56	×	0.3s	28

elimination in $\exists v' : next(v, v')$ can be automated for the theory of linear arithmetic. For dealing with well-foundedness we use linear ranking functions, and hence corresponding linear templates for DecreaseT and BoundT.

We report the results in Table 1. Columns 3 and 6 show ✓ marks for the cases where E-HSF was able to find a solution, i.e., prove the CTL property. See Columns 4 and 7 for the time spent on finding solutions. E-HSF is able to find proofs for all the correct programs except for P14 and P15 that correspond to Windows frag.4. Currently, E-HSF models the control flow symbolically using a program counter variable, which is most likely the reason for not succeeding on P14 and P15. Efficient treatment of control flow along the lines of explicit analysis as performed in the CPAchecker framework could lead to significant improvements for dealing with programs with large control-flow graphs [4].

For cases where the property contains more than one path quantifier and the top-most temporal quantifier is F or U, our implementation generates non-Horn clauses following the proof system from [25]. While a general algorithm for solving non-Horn clauses is beyond the scope of this paper, we used a simple heuristic to seed solutions for queries appearing under the negation operator. For example, for the verification task obtained from proving ϕ for P2, we used the solution $(a = 1 \wedge r \neq 5)$ for the query corresponding to the nesting structure of ϕ. This solution is obtained as a conjunction of the atomic constraints from ϕ.

7 Related Work

Our work is inspired by a recent approach to CTL verification of programs [11]. The main similarity lies in the use of a refinement loop to discover witnesses for resolving non-determinism/existentially quantified variables. The main difference lies in the way candidate witnesses are selected. While [11] refines witnesses, i.e., the non-determinism in witness relations monotonically decreases at each iteration, E-HSF can change witness candidates arbitrarily (yet, subject to the global constraint). Thus, our method can backtrack from wrong choices in cases when [11] needs to give up.

E-HSF generalizes solving methods for universally quantified Horn clauses over decidable theories, e.g. [19, 23, 27]. Our approach relies on the templates for describing the space of candidate witnesses. Computing witnesses using a generalisation approach akin to PDR [23] is an interesting alternative to explore in future work.

Template based synthesis of invariants and ranking functions is a prominent technique for dealing with universal properties, see e.g. [10, 21, 30, 33]. E-HSF implementation of EncodeValidity supporting linear arithmetic inequalities is directly inspired by these techniques, and puts them to work for existential properties.

Decision procedures for quantified propositional formulas on bit as well as word level [24,34] rely on iteration and refinement for the discovery of witnesses. The possibility of integration of QBF solvers as an implementation of EncodeValidity is an interesting avenue for future research.

Some formulations of proof systems for mu-calculus, e.g., [14] and [28], could be seen as another source of forall-exists clauses (to pass to E-HSF). Compared to the XSB system [14] that focuses on finite state systems, E-HSF aims at infinite state systems and employs a CEGAR-based algorithm. XSB's extensions for infinite state systems are rather specific, e.g., data-independent systems, and do not employ abstraction refinement techniques. Finally, we remark that abstraction-based methods, like ours, can be complemented with program specialization-based methods for verification of CTL properties [17].

8 Conclusion

Verification conditions for proving existential temporal properties of programs can be represented using existentially quantified Horn-like clauses. In this paper

we presented a counterexample guided method for solving such clauses, which can compute witnesses to existentially quantified variables in form of linear arithmetic expressions. By aggregating constraints on witness relations across different counterexamples our method can recover from wrong choices. We leave the evaluation of applicability of our method for other problems requiring witness computation, e.g., software synthesis or game solving to future work.

Acknowledgements. We thank Byron Cook and Eric Koskinen for valuable discussion and for generously making their benchmarks available. This research was supported in part by ERC project 308125 VeriSynth and by the DFG Graduiertenkolleg 1480 (PUMA).

References

1. Albarghouthi, A., Li, Y., Gurfinkel, A., Chechik, M.: UFO: A framework for abstraction- and interpolation-based software verification. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 672–678. Springer, Heidelberg (2012)
2. Ball, T., Rajamani, S.K.: The SLAM project: debugging system software via static analysis. In: POPL (2002)
3. Beyer, D., Keremoglu, M.E.: CPACHECKER: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
4. Beyer, D., Löwe, S.: Explicit-state software model checking based on CEGAR and interpolation. In: Cortellessa, V., Varró, D. (eds.) FASE 2013. LNCS, vol. 7793, pp. 146–162. Springer, Heidelberg (2013)
5. Blanchet, B., Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., Rival, X.: A static analyzer for large safety-critical software. In: PLDI (2003)
6. Bruttomesso, R., Cimatti, A., Franzén, A., Griggio, A., Sebastiani, R.: The MATHSAT 4 SMT solver. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 299–303. Springer, Heidelberg (2008)
7. Cadar, C., Dunbar, D., Engler, D.R.: KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In: OSDI (2008)
8. Clarke, E., Kroning, D., Lerda, F.: A tool for checking ANSI-C programs. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 168–176. Springer, Heidelberg (2004)
9. Clarke, E., Kroning, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 570–574. Springer, Heidelberg (2005)
10. Colón, M.A., Sankaranarayanan, S., Sipma, H.B.: Linear invariant generation using non-linear constraint solving. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 420–432. Springer, Heidelberg (2003)
11. Cook, B., Koskinen, E.: Reasoning about nondeterminism in programs. In: PLDI (2013)
12. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI (2006)
13. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL (1977)

14. Cui, B., Dong, Y., Du, X., Narayan Kumar, K., Ramakrishnan, C.R., Ramakrishnan, I.V., Roychoudhury, A., Smolka, S.A., Warren, D.S.: Logic programming and model checking. In: Palamidessi, C., Meinke, K., Glaser, H. (eds.) ALP 1998 and PLILP 1998. LNCS, vol. 1490, pp. 1–20. Springer, Heidelberg (1998)
15. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
16. Emmes, F., Enger, T., Giesl, J.: Proving non-looping non-termination automatically. In: Gramlich, B., Miller, D., Sattler, U. (eds.) IJCAR 2012. LNCS, vol. 7364, pp. 225–240. Springer, Heidelberg (2012)
17. Fioravanti, F., Pettorossi, A., Proietti, M., Senni, V.: Generalization strategies for the verification of infinite state systems. Theory and Practice of Logic Programming 13, 175–199 (2013)
18. Graf, S., Saïdi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, Springer, Heidelberg (1997)
19. Grebenshchikov, S., Lopes, N.P., Popeea, C., Rybalchenko, A.: Synthesizing software verifiers from proof rules. In: PLDI (2012)
20. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL (2008)
21. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 262–276. Springer, Heidelberg (2009)
22. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL (2004)
23. Hoder, K., Bjørner, N., de Moura, L.: μZ– an efficient engine for fixed points with constraints. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 457–462. Springer, Heidelberg (2011)
24. Janota, M., Klieber, W., Marques-Silva, J., Clarke, E.: Solving QBF with counterexample guided refinement. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 114–128. Springer, Heidelberg (2012)
25. Kesten, Y., Pnueli, A.: A compositional approach to CTL* verification. Theor. Comput. Sci. 331(2-3), 397–428 (2005)
26. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
27. McMillan, K.L., Rybalchenko, A.: Computing relational fixed points using interpolation. Technical report, available from authors (2012)
28. Namjoshi, K.S.: Certifying model checkers. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 2–13. Springer, Heidelberg (2001)
29. Payet, É., Spoto, F.: Experiments with non-termination analysis for Java Bytecode. Electr. Notes Theor. Comput. Sci. 253(5) (2009)
30. Podelski, A., Rybalchenko, A.: A complete method for the synthesis of linear ranking functions. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 239–251. Springer, Heidelberg (2004)
31. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS (2004)
32. Schrijver, A.: Theory of linear and integer programming. Wiley-Interscience series in discrete mathematics and optimization. Wiley (1999)
33. Srivastava, S., Gulwani, S.: Program verification using templates over predicate abstraction. In: PLDI (2009)
34. Wintersteiger, C.M., Hamadi, Y., de Moura, L.M.: Efficiently solving quantified bit-vector formulas. Formal Methods in System Design (2013)

GOAL for Games, Omega-Automata, and Logics

Ming-Hsien Tsai, Yih-Kuen Tsay, and Yu-Shiang Hwang

National Taiwan University

Abstract. This paper introduces the second generation of GOAL, which is a graphical interactive tool for games, ω-automata, and logics. It is a complete redesign with an extensible architecture, many enhancements to existing functions, and new features. The extensible architecture allows easy integration of third-party plugins. The enhancements provide more automata conversion, complementation, simplification, and testing algorithms, translation of full QPTL formulae, and better automata navigation with more layout algorithms and utility functions. The new features include game solving, manipulation of two-way alternating automata, translation of ACTL formulae and ω-regular expressions, test of language star-freeness, classification of ω-regular languages into the temporal hierarchy of Manna and Pnueli, and a script interpreter.

1 Introduction

GOAL (http://goal.im.ntu.edu.tw) is a graphical interactive tool for defining and manipulating games, ω-automata, and logic formulae. The first generation of GOAL (or simply the 1st gen) was formally introduced in [52] and later extended in [51]. It is implemented in Java and built upon the classic finite automata and graphical modules of JFLAP [40]. The main features of the 1st gen include (1) drawing games and ω-automata, (2) translating quantified propositional temporal logic (QPTL, an extension of PTL and also LTL) formulae in prenex normal form to equivalent Büchi automata, (3) complementing Büchi automata, (4) testing containment and equivalence between two Büchi automata, and (5) exporting Büchi automata as Promela code. It also provides a command-line mode and utility functions for collecting statistic data and generating random automata and temporal formulae.

A typical usage of GOAL is for educational purposes, helping the user get a better understanding of Büchi automata and their relation to temporal logic. For example, the user may follow a translation algorithm step-by-step in the tool to see how a QPTL formula is translated to an equivalent Büchi automaton. GOAL may also be used for supplementing automata-theoretic model checkers such as SPIN [20]. For example, the user may construct a smaller specification automaton that is checked to be correct in that it is equivalent to a larger reference Büchi automaton or a QPTL formula. Moreover, GOAL has been used for supporting research and tools development [1,3,8,50,53].

This paper introduces the second generation of GOAL, which is a complete redesign with an extensible architecture, many enhancements to existing functions, and new features. The extensible architecture allows easy integration of

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 883–889, 2013.

Table 1. Major algorithms in GOAL. An * indicates that the implementation of the algorithm has already been reported in [52,51].

Translation of QPTL Formulae
Tableau*, Incremental Tableau* [26], Temporal Tester* [27], GPVW* [16], GPVW+* [16], LTL2AUT* [16], LTL2AUT+* [49], LTL2BA* [14], PLTL2BA* [15], MoDeLLa [44,49], Couvreur's [10,49], LTL2BUCHI [17,49], KP02 [28], CCJ09 [9]
Complementation of Büchi Automata
Safra* [41], WAPA* [48], WAA* [30], Safra-Piterman* [39], Kurshan's [31], Ramsey-based [5,45], Muller-Schupp [37,2], Rank-based [30,43], Slice-based [25]
Simplification of Automata
Direct and Reverse Simulation* [47], Pruning Fair Sets* [47], Delayed Simulation [12], Fair Simulation [19], Parity Simplification [6]
Parity Game Solving
Recursive [32], McNaughton-Zielonka [36,54], Dominion Decomposition [24], Small Progress Measure [23], Big Steps [42], Global Optimization [13]

third-party plugins. The enhancements provide many more operations and tests on automata, more translation algorithms for QPTL (one of which supports full QPTL formulae that are not required to be in prenex normal form), and better automata navigation with more layout algorithms and utility functions. The new features provide game solving and conversion, support of two-way alternating automata and $\forall$ computation tree logic (ACTL), and more classification information about ω-regular languages such as star-freeness. Table 1 summarizes the major algorithms implemented in GOAL.

2 Enhanced Functions

We describe in this section how the functions of the 1st gen have been extended.

- **Translation of QPTL Formulae:** We have implemented five more translation algorithms for QPTL, namely KP02 [28], CCJ09 [9], MoDeLLa [44], Couvreur's [10], and LTL2BUCHI [17]. KP02 is the only one that supports full QPTL by an inductive construction, while CCJ09 can translate four specific patterns of QPTL formulae to minimal Büchi automata. We have also extended MoDeLLa, Couvreur's, and LTL2BUCHI to allow past operators.
- **Conversion between Automata:** Compared to the 1st gen, GOAL now supports many more conversions between automata, especially the conversion from nondeterministic Büchi automata, if possible, to deterministic automata with Büchi [33,4] or co-Büchi [4] acceptance conditions. Moreover, GOAL can automatically find and apply a sequence of chained conversions to convert an automaton to another type specified by the user.
- **Operation and Simplification on Automata:** GOAL has five more complementation constructions for Büchi automata, namely Kurshan's [31], Ramsey-based [5,45], Muller-Schupp [37,2], Rank-based [30,43], and Slice-based [25]. With chained conversions, several Boolean operations previously implemented only for Büchi automata can be applied to any automata

convertible to equivalent Büchi automata. We have also implemented simplification of Büchi automata with delayed simulation and fair simulation relations based on solving simulation games [12,19].
- **Test on Automata:** GOAL has five more containment testing algorithms, which are performed incrementally with a double depth-first search. Four of them are based on determinization-based complementation constructions [41,37,2,39], while one is based on slice-based constructions [25]. Such incremental algorithms usually can find counterexamples earlier. Same as in automata conversion, several tests implemented for Büchi automata can also be applied to other types of automata.
- **Automata Navigation:** Navigation of a large automaton can be cumbersome. To make it easier, GOAL has eleven new layout algorithms. After an automatic layout, the user can manually arrange the states with the help of guidelines, gridlines, and snapping to grids to make the layout even better. GOAL also allows the user to focus on a state and its neighbors such that the user can easily traverse a particular path of an automaton.

3 New Features

We now detail the new features of GOAL.

- **Extensibility:** GOAL can be easily extended with the help of Java Plugin Framework [22]. The user may add a new menu item, command, or algorithm with a plugin simply by extending the classes or interfaces of GOAL and then writing a configuration file for the plugin. During runtime, GOAL will read the configuration file and enable the third-party plugin. In fact, GOAL itself is composed of three plugins, namely CORE, UI, and CMD, where CORE provides basic data structures and implementations of algorithms, UI provides a graphical interface, and CMD provides a command-line interface.
- **Game Solving and Conversion:** We have implemented eight game solving algorithms of which one is for reachability games, one for Büchi games, and six for parity games. Winning regions and strategies in a solved game are highlighted and can be exported with the game to a file. Several conversions between games, including the conversion from a Muller game to a parity game [35], are implemented. To help experiments with games, the generation of random games is provided as well. GOAL can also take the product of a game arena (that describes the allowed interactions between a module and its environment) and a specification automaton (resulting in a game), and hence may be used to experiment with the synthesis process in a game-based approach to the synthesis of reactive modules [46].
- **New Types of Automata and Logics:** Previously in the 1st gen, two-way alternating automata were only used internally in PLTL2BA. Now the user can draw a two-way alternating automaton and convert it to an equivalent transition-based generalized Büchi automaton if it is very weak [15]. For new logics, GOAL supports the translation of an ACTL formula [38] to a

maximal model (represented as an automaton) of the formula [18,29]. Such model can be used in model checking or synthesis. The translation of ω-regular expressions is also available in GOAL.

- **Classification of ω-Regular Languages:** We have implemented an algorithm for testing whether an ω-regular language or Büchi automaton is star-free [11]. If an ω-regular language is star-free, it can be specified by a formula in PTL, which is less expressive than QPTL. We have also implemented the classification of ω-regular languages into the temporal hierarchy of Manna and Pnueli [34]. Such classification can be used not only for educational purposes but also for helping model checking [7].
- **Script Interpreter:** GOAL provides an interpreter that can execute scripts in a customized language. In the 1st gen, GOAL commands can only be executed in shell scripts, which create a GOAL process per command. Now a batch of GOAL commands can be written as a script and executed by a single GOAL process to achieve better performance.

4 Remarks

With these enhancements and new features, GOAL now lives up to an alternative source of its acronym "**G**ames, **O**mega-**A**utomata, and **L**ogics". Even classic finite automata and regular expressions are also supported by GOAL. With the new architecture, we expect that GOAL will be extended by third-party plugins. We will also continue to extend GOAL with more algorithms, e.g., translation algorithms for Property Specification Language (PSL) [21] and game solving algorithms that produce better winning strategies.

Acknowledgements. This work was partially supported by the National Science Council, Taiwan, under grants NSC97-2221-E-002-074-MY3, NSC100-2221-E-002-116, and NSC101-2221-E-002-093. We thank Jinn-Shu Chang and Yi-Wen Chang for helping with the implementations of some algorithms.

References

1. Abdulla, P.A., Chen, Y.-F., Clemente, L., Holík, L., Hong, C.-D., Mayr, R., Vojnar, T.: Simulation subsumption in Ramsey-based Büchi automata universality and inclusion testing. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 132–147. Springer, Heidelberg (2010)
2. Althoff, C.S., Thomas, W., Wallmeier, N.: Observations on determinization of Büchi automata. TCS 363(2), 224–233 (2006)
3. Alur, R., Weiss, G.: RTComposer: a framework for real-time components with scheduling interfaces. In: EMSOFT, pp. 159–168. ACM (2008)
4. Boker, U., Kupferman, O.: Co-ing Büchi made tight and useful. In: LICS, pp. 245–254. IEEE Computer Society (2009)
5. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: CLMPS, pp. 1–12. Stanford University Press (1962)

6. Carton, O., Maceiras, R.: Computing the Rabin index of a parity automaton. Informatique Théorique et Applications 33(6), 495–506 (1999)
7. Černá, I., Pelánek, R.: Relating hierarchy of temporal properties to model checking. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 318–327. Springer, Heidelberg (2003)
8. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Singh, R.: QUASY: Quantitative synthesis tool. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 267–271. Springer, Heidelberg (2011)
9. Cichoń, J., Czubak, A., Jasiński, A.: Minimal Büchi automata for certain classes of LTL formulas. In: DepCos-RELCOMEX, pp. 17–24. IEEE (2009)
10. Couvreur, J.-M.: On-the-fly verification of linear temporal logic. In: Wing, J.M., Woodcock, J. (eds.) FM 1999. LNCS, vol. 1708, pp. 253–271. Springer, Heidelberg (1999)
11. Diekert, V., Gastin, P.: First-order definable languages. In: Logic and Automata. Texts in Logic and Games, vol. 2, pp. 261–306. Amsterdam Univ. Press (2008)
12. Etessami, K., Wilke, T., Schuller, R.A.: Fair simulation relations, parity games, and state space reduction for Büchi automata. SICOMP 34(5), 1159–1175 (2005)
13. Friedmann, O., Lange, M.: Solving parity games in practice. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 182–196. Springer, Heidelberg (2009)
14. Gastin, P., Oddoux, D.: Fast LTL to Büchi automata translation. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 53–65. Springer, Heidelberg (2001)
15. Gastin, P., Oddoux, D.: LTL with past and two-way very-weak alternating automata. In: Rovan, B., Vojtáš, P. (eds.) MFCS 2003. LNCS, vol. 2747, pp. 439–448. Springer, Heidelberg (2003)
16. Gerth, R., Peled, D., Vardi, M.Y., Wolper, P.: Simple on-the-fly automatic verification of linear temporal logic. In: PSTV, pp. 3–18. Chapman & Hall (1995)
17. Giannakopoulou, D., Lerda, F.: From states to transitions: Improving translation of LTL formulae to Büchi automata. In: Peled, D.A., Vardi, M.Y. (eds.) FORTE 2002. LNCS, vol. 2529, pp. 308–326. Springer, Heidelberg (2002)
18. Grumberg, O., Long, D.E.: Model checking and modular verification. TOPLAS 16(3), 843–871 (1994)
19. Gurumurthy, S., Bloem, R., Somenzi, F.: Fair simulation minimization. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 610–624. Springer, Heidelberg (2002)
20. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (September 2003)
21. IEEE standard for property specification language (PSL). IEEE Std 1850-2010 (Revision of IEEE Std 1850-2005), 1–182 (2010)
22. Java Plugin Framework, `http://jpf.sourceforge.net`
23. Jurdziński, M.: Small progress measures for solving parity games. In: Reichel, H., Tison, S. (eds.) STACS 2000. LNCS, vol. 1770, pp. 290–301. Springer, Heidelberg (2000)
24. Jurdziński, M., Paterson, M., Zwick, U.: A deterministic subexponential algorithm for solving parity games. SICOMP 38(4), 1519–1532 (2008)
25. Kähler, D., Wilke, T.: Complementation, disambiguation, and determinization of Büchi automata unified. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part I. LNCS, vol. 5125, pp. 724–735. Springer, Heidelberg (2008)

26. Kesten, Y., Manna, Z., McGuire, H., Pnueli, A.: A decision algorithm for full propositional temporal logic. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 97–109. Springer, Heidelberg (1993)
27. Kesten, Y., Pnueli, A.: Verification by augmented finitary abstraction. Information and Computation 163(1), 203–243 (2000)
28. Kesten, Y., Pnueli, A.: Complete proof system for QPTL. Journal of Logic and Computation 12(5), 701–745 (2002)
29. Klotz, T., Seßler, N., Straube, B., Fordran, E.: Compositional verification of material handling systems. In: ETFA, pp. 1–8. IEEE (2012)
30. Kupferman, O., Vardi, M.Y.: Weak alternating automata are not that weak. TOCL 2(3), 408–429 (2001)
31. Kurshan, R.P.: Complementing deterministic Büchi automata in polynomial time. JCSS 35(1), 59–71 (1987)
32. Küsters, R.: Memoryless determinacy of parity games. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 95–106. Springer, Heidelberg (2002)
33. Landweber, L.H.: Decision problems for omega-automata. Mathematical Systems Theory 3(4), 376–384 (1969)
34. Manna, Z., Pnueli, A.: A hierarchy of temporal properties. In: PODC, pp. 377–410 (1990)
35. Mazala, R.: Infinite games. In: Grädel, E., Thomas, W., Wilke, T. (eds.) Automata, Logics, and Infinite Games. LNCS, vol. 2500, pp. 23–38. Springer, Heidelberg (2002)
36. McNaughton, R.: Infinite games played on finite graphs. Annals of Pure and Applied Logic 65(2), 149–184 (1993)
37. Muller, D.E., Schupp, P.E.: Simulating alternating tree automata by nondeterministic automata: New results and new proofs of the theorems of Rabin, McNaughton and Safra. TCS 141(1&2), 69–107 (1995)
38. Peng, H., Mokhtari, Y., Tahar, S.: Environment synthesis for compositional model checking. In: ICCD, pp. 70–75. IEEE Computer Society (2002)
39. Piterman, N.: From nondeterministic Büchi and Streett automata to deterministic parity automata. LMCS 3(3), paper 5 (2007)
40. Rodger, S., Finley, T.: JFLAP, `http://www.jflap.org/`
41. Safra, S.: On the complexity of ω-automata. In: FOCS, pp. 319–327. IEEE (1988)
42. Schewe, S.: Solving parity games in big steps. In: Arvind, V., Prasad, S. (eds.) FSTTCS 2007. LNCS, vol. 4855, pp. 449–460. Springer, Heidelberg (2007)
43. Schewe, S.: Büchi complementation made tight. In: STACS. LIPIcs, vol. 3, pp. 661–672 (2009)
44. Sebastiani, R., Tonetta, S.: "More" deterministic vs. "smaller" Büchi automata for efficient LTL model checking. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 126–140. Springer, Heidelberg (2003)
45. Sistla, A.P., Vardi, M.Y., Wolper, P.: The complementation problem for Büchi automata with appplications to temporal logic. TCS 49, 217–237 (1987)
46. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, pp. 77–84. IEEE (2009)
47. Somenzi, F., Bloem, R.: Efficient Büchi automata from LTL formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 248–263. Springer, Heidelberg (2000)
48. Thomas, W.: Complementation of Büchi automata revisited. In: Jewels are Forever, pp. 109–120. Springer (1999)

49. Tsai, M.-H., Chan, W.-C., Tsay, Y.-K., Luo, C.-J.: Incremental translation of full PTL formulae to Büchi automata (manuscript 2013)
50. Tsai, M.-H., Fogarty, S., Vardi, M.Y., Tsay, Y.-K.: State of Büchi complementation. In: Domaratzki, M., Salomaa, K. (eds.) CIAA 2010. LNCS, vol. 6482, pp. 261–271. Springer, Heidelberg (2011)
51. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Chan, W.-C., Luo, C.-J.: GOAL extended: Towards a research tool for omega automata and temporal logic. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 346–350. Springer, Heidelberg (2008)
52. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Wu, K.-N., Chan, W.-C.: GOAL: A graphical tool for manipulating Büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 466–471. Springer, Heidelberg (2007)
53. Tsay, Y.-K., Tsai, M.-H., Chang, J.-S., Chang, Y.-W., Liu, C.-S.: Büchi Store: An open repository of ω-automata. STTT 15(2), 109–123 (2013)
54. Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. TCS 200(1-2), 135–183 (1998)

PRALINE: A Tool for Computing Nash Equilibria in Concurrent Games*

Romain Brenguier

Département d'informatique, Université Libre de Bruxelles (U.L.B), Belgium
LSV, CNRS & ENS Cachan, France
brenguier@lsv.ens-cachan.fr

Abstract. We present PRALINE, which is the first tool to compute Nash equilibria in games played over graphs. We consider concurrent games: at each step, players choose their actions independently. There can be an arbitrary number of players. The preferences of the players are given by payoff functions that map states to integers, the goal for a player is then to maximize the limit superior of her payoff; this can be seen as a generalization of Büchi objectives. PRALINE looks for pure Nash equilibria in these games. It can construct the strategies of the equilibrium and users can play against it to test the equilibrium. We give the idea behind its implementation and present examples of its practical use.

1 Introduction

In computer science, two-player games have been successfully used for solving the problem of controller synthesis. Multiplayer games appear when we model interaction between several agents, each agent having its own preference concerning the evolution of the global system. Think for instance of users behind their computers on a network. When designing a protocol, maximizing the overall performance of the system is desirable, but if a deviation can be profitable to one user, it should be expected from him to take advantage of this weakness. This happened for example to the bit-torrent protocol where selfish clients became more popular. Such deviations can harm the global performance of the protocol. In these situations, equilibrium concepts are particularly relevant. They aim at describing rational behaviors. In a Nash equilibrium, each agent plays in such a way that none of them can get a better payoff by switching to another strategy.

In the context of controller synthesis, games are generally played on graphs. The nodes of the graph represent the possible configurations of the system. The agents take actions in order to move a token from one node to another. Among those games, the simplest model is that of turn-based games, where in each state, one player decides alone on which outgoing edge to take. The model we consider is concurrent games, which is more expressive. For these games, in each state, the players choose their actions independently and the joint move formed by these choices determines the next state.

* Work supported by ERC Starting Grant inVEST (279499).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 890–895, 2013.

There has recently been a lot of focus on the algorithmic aspect of Nash equilibria for games played on graphs [4,11,12]. Thanks to these efforts, the theoretical bases are understood well enough, so that we have developed effective algorithms [1,2]. We implemented them in PRALINE. Other tools exist which can compute Nash equilibria for classical models, however this is the first tool to compute Nash equilibria in games played on graphs. The tool PRISM-games [5] can also analyse multiplayer games on graphs. In particular, it can generate an optimal strategy for one player and can be used to check that a given strategy is a Nash equilibrium. It can deal with randomized games, which PRALINE cannot, however it is unable to generate Nash equilibria, as PRALINE does.

We give an overview of the features of PRALINE. First, we present the model of games that is used, illustrated with examples. We also present the suspect game transformation [3], which gives the idea of the underlying algorithm and makes it possible to test the equilibrium by playing against it. We also ran some experiments on the given examples to evaluate the performances of the tool. The tool is available from `http://www.lsv.ens-cachan.fr/Software/praline/`.

2 Concurrent Games

The model of game we consider is concurrent games. These are played on a graph which we call the *arena* of the game. A *state* of the game is a vertex of the arena. At the beginning of a turn, each of the players chooses an action, and the tuple of these actions defines a *move*. The next state of the game is given by following the edge that goes from the current state and is labeled by this move, and a new turn begins from that state. This is then repeated *ad infinitum*, to define an infinite path on the graph, called a *play*. Players are assumed to see the sequence of states, but not the actions played by other players.

An example of an arena is given in Fig. 2. If, for example, in state `1,0,1,0`, player p_1 chooses action 1 and player p_2 chooses action 0, the play follows the edge labeled by the move $1, 0$ and the next state is `0,1,1,0`. Then a new turn begins. If both players keep on playing $0, 0$ forever, the system will stay in configuration `0,1,1,0`; this defines the play $\texttt{1,0,1,0} \cdot (\texttt{0,1,1,0})^{\omega}$.

To describe games with a big state space it is convenient to write small programs which generate the arena, like the one in Fig. 1. Each state of the arena corresponds to a possible valuation of the variables, the move function describes the actions available in each state and the update function computes the new state according to the actions of each player.

Example 1 (Medium Access Control). This example was first formalized from the point of view of game theory in [8]. Several users share access to a wireless channel. During each slot, they can choose to either transmit or wait for the next slot. If too many users are emitting in the same slot, then they fail to send data. Furthermore each attempt at transmitting costs energy to the players. They have to maximize the number of successful attempts using the energy available to them. We give in Fig. 1 a possible way to model this game in PRALINE. In this example the players are `p1` and `p2`. Their energy levels are represented by

variables `energy1` and `energy2`, and variables `trans1` and `trans2` keep track of the number of successful attempts. The players can always wait (represented by the action 0), and if there energy is not zero they can transmit (represented by action 1). The generated arena for an initial energy allowing only one attempt for each player is represented in Fig. 2. The labels of the nodes correspond to the valuation of the variables `energy1`, `trans1`, `energy2` and `trans2`.

```
move {
  legal p1 0;
  legal p2 0;
  if (energy1 > 0) legal p1 1;
  if (energy2 > 0) legal p2 1;
}

update {
  if (action p1 == 1)
    energy1 = energy1 - 1;
  if (action p2 == 1)
    energy2 = energy2 - 1;

  if (action p1 == 1 && action p2 == 0)
    trans1 = trans1 + 1;
  if (action p1 == 0 && action p2 == 1)
    trans2 = trans2 + 1;
}
```

Fig. 1. Part of the game file "medium_access.game"

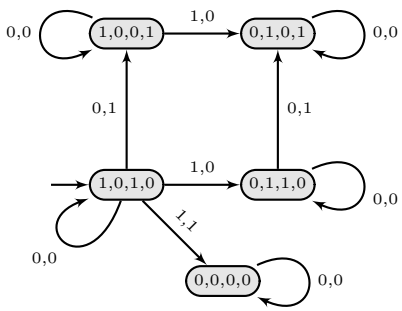


Fig. 2. Arena generated from the file "medium_access.game"

3 Computing Nash Equilibria

The preference of a player p_i is specified by a function payoff_{p_i} which assigns an integer to each state of the game. The payoff of a run is the limit superior of this function, and the goal is to maximize it. This is a generalization of Büchi objectives which can be specified with payoff either 0 or 1 for each state.

Example 2 (Power Control). This example is inspired by the problem of power control in cellular networks. Game theoretical concepts are relevant for this problem and Nash equilibria are actually used to describe rational behaviors of agents [6,7]. We consider the situation where several phones are emitting over a cellular network. Each agent p_i can choose the emitting power pow_i of his phone. From the point of view of agent p_i, using a stronger power results in a better transmission, but it is costly since it uses energy, and it lowers the quality of the transmission for the others, because of interferences. We model this game by the arena presented in Fig. 3, for a simple situation with two players which at each step can choose to increase or not their emitting power until they reach the maximum level of 2. The payoff for player p_i can be modeled by this expression from [10]: $\mathsf{payoff}_{p_i} = \frac{R}{\mathsf{pow}_i}\left(1 - e^{-0.5\gamma_i}\right)^L$ where γ_i is the signal-to-interference-and-noise ratio for player p_i, R is the rate at which the wireless system transmits the information in bits per seconds and L is the size of the packets in bits.

To describe the rational behavior of the agents in a non-zero-sum game, the concept that is most commonly used is Nash equilibria [9]: a Nash equilibrium is a choice of strategies (one for each player), such that there is no player which can increase her payoff, by changing her own strategy, while the other players keep theirs unchanged.

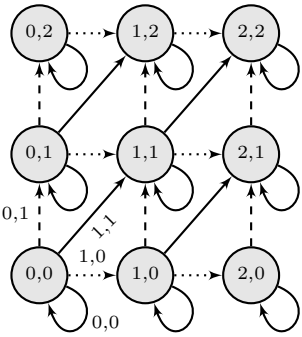


Fig. 3. Arena of the power control game

The tool PRALINE looks for pure (i.e. non randomized) Nash equilibria in the kind of games we described. Note that a pure Nash equilibrium is resistant to randomized strategies. On the other hand the existence of a randomized Nash equilibrium with a particular payoff is undecidable [12]. If the game contains a (pure) Nash equilibrium with some payoff payoff_i for each player p_i, then PRALINE returns at least one Nash equilibrium with payoff payoff'_i such that for every player p_i, $\mathsf{payoff}'_i \geq \mathsf{payoff}_i$. For an overview of the Nash equilibrium, the tool can output the *shape* of the solution. That is, the moves that are effectively taken by the players if none of them deviates from the equilibrium. For example, in the power control game, our tool gives two solutions with different payoffs, their shapes are represented in Fig. 4 and Fig. 5 respectively. For each solution, the tool can also output a file containing the full strategies, represented as automata. These automata are usually big: the first solution of the power control example is implemented by an automaton containing 125 edges. They can be difficult to analyze. A convenient way to look at the generated equilibrium, is to play the suspect game against it. We now explain the suspect game construction, which is the core of the implemented algorithm.

4 The Suspect Game

The idea behind the algorithm implemented is that of the suspect game, which allows to think about Nash equilibria as winning strategies in a two-player turn-based game [3]. This transformation makes it possible to use algorithmic techniques from zero-sum games to synthesize Nash equilibria.

The suspect game is played between `Eve` and `Adam`. `Eve` wants the players to play a Nash equilibrium, and `Adam` tries to disprove that it is a Nash equilibrium, by finding a possible deviation that improves the payoff of one of the players. In the beginning of the game all the players are considered suspect, since they can potentially deviate from the equilibrium. Then `Eve` chooses a legal move

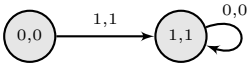


Fig. 4. Shape of solution 1

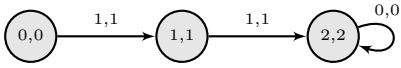


Fig. 5. Shape of solution 2

and Adam chooses some successor state. When the state chosen by Adam is the state resulting from Eve's move, we say that Adam obeys Eve, and the suspects are the same than before. Otherwise we keep among the suspects those that can unilaterally change their action from the one suggested by Eve to activate the transition suggested by Adam. The game then continues from the state played by Adam. Given the desired payoff, the outcome of the game is winning for Eve, if all players that are ultimately suspect have a payoff inferior or equal to the given one. There is a Nash equilibrium in the original game with payoff_i for each player p_i if, and only if, there is a winning strategy for Eve such that when Adam obeys the payoff is exactly payoff_i for each player p_i [3].

Using this transformation for the preferences under consideration, Eve's objective can be expressed as a co-Büchi condition. This game can be solved in polynomial time and we have indeed a polynomial time algorithm for Nash equilibria [2]. In order to understand how a Nash equilibrium is enforced, the tool PRALINE allows the user to play against the winning strategy of Eve.

Example 2 (cont'd). We come back to the example of the power control game, and the first solution, where players use a power of 1 and get a payoff of 110. When we play against Eve in this game, she first suggests the move $1, 1$. If we obey this move, the power of each players is raised to 1. Eve then plays $0, 0$ as long as we stay in this same state. If we continue to obey, the payoff will be 110 for each player, which would make Eve win. If we want to find a deviation profitable to one player we might want to raise the power of one of them. For instance, if we change the power of the first one, then she is suspect in the next state and her current payoff is 131. But then, Eve will suggest the move $0, 1$ whose natural outcome is $2, 2$ which has a payoff of 94 for the first player. If we obey this move we failed to improve our payoff, and we cannot change the next state by changing only the action of the first player, so the game is lost for Adam.

5 Experiments and Conclusions

In order to show the influence of the size of the graph on the time taken to compute Nash equilibria, we ran the tool on examples with different parameters. The experimental results are given in Table 1, they were obtained on a PC with an Intel Core2 Duo processor at 2.8GHz with 4GB of RAM. We observe from these experiments that our prototype works well for games up to one hundred states. The execution time then quickly increases. This is because the algorithm as described in [2], requires computation of the winning regions in a number of subgames that might be quadratic in the number of states of the game. Computing winning regions takes quadratic time in itself.

PRALINE is the first tool to compute pure Nash equilibria in games played on graphs. Experimental results are encouraging since we managed to synthesize Nash equilibria for several examples. For future implementations, we hope to improve the tool's scalability by using symbolic methods.

Table 1. Experiments

Power Control					
Players	Emission Levels	States	Edges	Solutions	Time (sec.)
2	2	9	25	2	0.01
4	2	81	625	6	4.28
3	5	216	1331	83	64.50
6	2	729	15625	23	2700.97
Medium Access Control					
Players	Initial Energy	States	Edges	Solutions	Time (sec.)
2	2	14	35	1	0.02
3	2	100	347	1	0.69
3	4	1360	6303	1	160.22

References

1. Bouyer, P., Brenguier, R., Markey, N.: Nash equilibria for reachability objectives in multi-player timed games. In: Gastin, P., Laroussinie, F. (eds.) CONCUR 2010. LNCS, vol. 6269, pp. 192–206. Springer, Heidelberg (2010)
2. Bouyer, P., Brenguier, R., Markey, N., Ummels, M.: Nash equilibria in concurrent games with Büchi objectives. In: FSTTCS 2011, pp. 375–386. Leibniz-Zentrum für Informatik (2011)
3. Bouyer, P., Brenguier, R., Markey, N., Ummels, M.: Concurrent games with ordered objectives. In: Birkedal, L. (ed.) FOSSACS 2012. LNCS, vol. 7213, pp. 301–315. Springer, Heidelberg (2012)
4. Chatterjee, K.: Two-player nonzero-sum ω-regular games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 413–427. Springer, Heidelberg (2005)
5. Chen, T., Forejt, V., Kwiatkowska, M., Parker, D., Simaitis, A.: PRISM-games: A model checker for stochastic multi-player games. In: Piterman, N., Smolka, S.A. (eds.) TACAS 2013. LNCS, vol. 7795, pp. 185–191. Springer, Heidelberg (2013)
6. MacKenzie, A., Wicker, S.: Game theory and the design of self-configuring, adaptive wireless networks. IEEE Communications Magazine 39(11), 126–131 (2001)
7. MacKenzie, A., Wicker, S.: Game theory in communications: Motivation, explanation, and application to power control. In: GLOBECOM 2001, pp. 821–826. IEEE (2001)
8. MacKenzie, A., Wicker, S.: Stability of multipacket slotted aloha with selfish users and perfect information. IEEE INFOCOM 3, 1583–1590 (2003)
9. Nash Jr., J.F.: Equilibrium points in n-person games. Proc. National Academy of Sciences of the USA 36(1), 48–49 (1950)
10. Saraydar, C., Mandayam, N., Goodman, D.: Pareto efficiency of pricing-based power control in wireless data networks. In: 1999 IEEE Wireless Communications and Networking Conference, WCNC, pp. 231–235. IEEE (1999)
11. Ummels, M.: The complexity of Nash equilibria in infinite multiplayer games. In: Amadio, R.M. (ed.) FOSSACS 2008. LNCS, vol. 4962, pp. 20–34. Springer, Heidelberg (2008)
12. Ummels, M., Wojtczak, D.: The complexity of Nash equilibria in limit-average games. In: Katoen, J.-P., König, B. (eds.) CONCUR 2011. LNCS, vol. 6901, pp. 482–496. Springer, Heidelberg (2011)

Program Repair without Regret

Christian von Essen[1] and Barbara Jobstmann[1,2,3]

[1] Verimag, CNRS and Universities of Grenoble, France
[2] Jasper Design Automation, CA
[3] École Polytechnique Fédérale de Lausanne, Switzerland

Abstract. We present a new and flexible approach to repair reactive programs with respect to a specification. The specification is given in linear-temporal logic. Like in previous approaches, we aim for a repaired program that satisfies the specification and is syntactically close to the faulty program. The novelty of our approach is that it produces a program that is also semantically close to the original program by enforcing that a subset of the original traces is preserved. Intuitively, the faulty program is considered to be a part of the specification, which enables us to synthesize meaningful repairs, even for incomplete specifications.

Our approach is based on synthesizing a program with a set of behaviors that stay within a lower and an upper bound. We provide an algorithm to decide if a program is repairable with respect to our new notion, and synthesize a repair if one exists. We analyze several ways to choose the set of traces to leave intact and show the boundaries they impose on repairability. We have evaluated the approach on several examples.

1 Introduction

Writing a program that satisfies a given specification usually involves several rounds of debugging. Debugging a program is often a difficult and tedious task: the programmer has to find the bug, localize the cause, and repair it. Model checking [9, 26] has been successfully used to expose bugs in a program. There are several approaches [1, 8, 12, 15, 17, 28, 29, 37] to automatically find the possible location of an error. We are interested in automatically repairing a program. Automatic program repair takes a program and a specification and searches for a correct program that satisfies the specification and is syntactically close to the original program (cf. [2, 4, 5, 11, 14, 16, 18, 31, 35]). Existing approaches follow the same idea: first, introduce freedom into the program (e.g., by describing valid edits to the program), and then search for a way of resolving this freedom such that the modified program satisfies the specification or the given test cases. While these approaches have been shown very effective, they suffer from a common weakness: they give little or no guarantees on preserving correct behaviors (i.e., program behaviors that do not violate the specification). Therefore, a user of a repair procedure may later *regret* having applied a fix to a program because it introduced new bugs by modifying behaviors that are not explicitly specified or for which no test case is available. The approach presented by Chandra et al. [4] provides some guarantees by requiring that a valid repair needs to pass a set of positive test cases. Correct behaviors outside these test cases are left unconstrained and the repair can thus change them unpredictably.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 896–911, 2013.

We present the first repair approach that constructs repairs that are guaranteed to satisfy the specification and that are not only syntactically, but also semantically close to the original program. The key benefits of our approach are: (i) it maintains correct program behavior, (ii) it is robust w.r.t. generous program modifications, i.e., it does not produce degenerated programs if given too much freedom in modifying the program, (iii) it works well with incomplete specifications, because it considers the faulty program as part of the specification and preserves its core behavior, and finally (iv) it is easy to implement on top of existing technology. We believe that our framework will prove useful because it does not require a complete specification by taking the program as part of the specification. It therefore makes writing specifications for programs easier. Furthermore, specifications are often given as conjunctions of smaller specifications that are verified individually. In order to keep desired behaviors, classical repair approaches repair a program with respect to the entire specification. Our approach can provide meaningful repair suggestions while focusing only on parts of the specification.

Contributions. We present an example motivating the need for a new definition of program repair (Section 3). We define a new notion of repair for reactive programs and present an algorithm to compute such repairs (Section 4). The algorithm is based on synthesizing repairs with respect to a lower and an upper bound on the set of generated traces. We show the limitations of any repair approach that is based on preserving part of the program's behavior (Section 5). Finally, we present experimental results based on a prototype employing the NuSMV [7] model checker.

2 Preliminaries

Words, Languages, Alphabet Restriction and Extension. Let AP be the finite set of *atomic propositions*. We define the *alphabet* over AP (denoted Σ_{AP}) as the set of all evaluations of AP, i.e., $\Sigma_{\mathrm{AP}} = 2^{\mathrm{AP}}$. If AP is clear from the context or not relevant, then we omit the subscript in Σ_{AP}. A *word* w is an infinite sequence of letters from Σ. We use Σ^ω to denote the set of all words. A *language* L is a set of words, i.e., $\mathrm{L} \subseteq \Sigma^\omega$. Given a word $\mathrm{w} \in \Sigma^\omega$, we denote the letter at position i by w_i, where w_0 is the first letter. We use $\mathrm{w}_{..i}$ to denote the prefix of w up to position i, and $\mathrm{w}_{i..}$ to denote the suffix of w starting at position i. Given a set of propositions $\mathrm{I} \subseteq \mathrm{AP}$, we define the I-*restriction* of a word $\mathrm{w} \in \Sigma_{\mathrm{AP}}^\omega$, denoted by $\mathrm{w}{\downarrow_\mathrm{I}}$, as $\mathrm{w}{\downarrow_\mathrm{I}} = l_0 l_1 \cdots \in \Sigma_\mathrm{I}^\omega$ with $l_i = (\mathrm{w}_i \cap \mathrm{I})$ for all $i \geq 0$. Given a language $\mathrm{L} \subseteq \Sigma_{\mathrm{AP}}^\omega$ and a set $\mathrm{I} \subseteq \mathrm{AP}$, we define the I-*restriction* of L, denoted by $\mathrm{L}{\downarrow_\mathrm{I}}$, as the set of I-restrictions of all the words in L, i.e., $\mathrm{L}{\downarrow_\mathrm{I}} = \{\mathrm{w}{\downarrow_\mathrm{I}} \mid \mathrm{w} \in \mathrm{L}\}$. Given a word $w \in \Sigma_\mathrm{I}^\omega$ over a set of propositions $\mathrm{I} \subseteq \mathrm{AP}$, we use $w{\uparrow_{AP}}$ to denote the *extension* of w to the alphabet Σ_{AP}, i.e., $w{\uparrow_{AP}} = \{w' \in \Sigma_{\mathrm{AP}}^\omega \mid w'{\downarrow_I} = w\}$. Extension of a language $\mathrm{L} \subseteq \Sigma_\mathrm{I}^\omega$ is defined analogously, i.e., $\mathrm{L}{\uparrow_{AP}} = \{w{\uparrow_{AP}} \mid w \in \mathrm{L}\}$. A language $\mathrm{L} \subseteq \Sigma_{\mathrm{AP}}^\omega$ is called I-*deterministic* for some set $\mathrm{I} \subseteq \mathrm{AP}$ if for each word $v \in \Sigma_\mathrm{I}^\omega$ there is at most one word $w \in \mathrm{L}$ such that $w{\downarrow_\mathrm{I}} = v$. A language L is called I-*complete* if for each input word $v \in \Sigma_\mathrm{I}^\omega$ there exists at least one word $w \in \mathrm{L}$ such that $w{\downarrow_I} = v$.

Machines, Automata, and Formulas. A *(finite state) machine* is a tuple $\mathrm{M} = (\mathrm{Q}, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \mathrm{q}_0, \delta, \gamma)$, where Q is a finite set of *states*, $\Sigma_\mathrm{I}(= 2^I)$ and $\Sigma_\mathrm{O}(= 2^O)$ are the *input* and the *output alphabet*, respectively, $\mathrm{q}_0 \in Q$ is the *initial state*, $\delta : Q \times \Sigma_\mathrm{I} \to Q$ is the *transition function*, and $\gamma : Q \times \Sigma_\mathrm{I} \to \Sigma_\mathrm{O}$ is the *output function*. The *input signals* I and the *output signals* O *of* M are required to be distinct, i.e., $I \cap O = \emptyset$. A *run* ρ of M

on an input word $\mathrm{w} \in {\Sigma_\mathrm{I}}^\omega$ is the sequence of states that the machine visits while reading the input word, i.e., $\rho = q_0 q_1 \cdots \in \mathrm{Q}^\omega$ such that $\delta(q_i, w_i) = q_{i+1}$ for all $i \geq 0$. The *output word* M produces on w (denoted by $\mathrm{M_O}(w)$) is the sequence of output letters that the machine produces while reading the input word, i.e., for the run $q_0 q_1 \ldots$ of M on w, the output word is $\mathrm{M_O}(w) = l_0 l_1 \cdots \in \Sigma_\mathrm{O}^\omega$ with $l_i = \gamma(q_i, \mathrm{w}_i)$ for all $i \geq 0$. The *combined input output word* M produces on w is defined as $\mathrm{M(w)} := (i_0 \cup o_0)(i_1 \cup o_1) \ldots \in \Sigma_\mathrm{AP}^\omega$, where $\mathrm{w} = i_0 i_1 \ldots$ and $\mathrm{M_O(w)} = o_0 o_1 \ldots$. We denote by $\mathrm{L(M)}$ the *language* of M, i.e., the set of combined input/output words $\mathrm{L(M)} = \{\mathrm{M(w)} \mid \mathrm{w} \in \Sigma_I^\omega\}$.

A *Büchi automaton* is a tuple $\mathrm{A} = (\mathrm{S}, \Sigma, \mathrm{s}_0, \Delta, \mathrm{F})$ where S is a finite set of *states*, Σ is the *alphabet*, $\mathrm{s}_0 \in \mathrm{S}$ is the *initial state*, $\Delta \subseteq \mathrm{S} \times \Sigma \times \mathrm{S}$ is the *transition relation*, and $\mathrm{F} \subseteq \mathrm{S}$ is the set of *accepting states*. A run of A on a word $\mathrm{w} \in \Sigma^\omega$ is a sequence of states $\mathrm{s}_0\, \mathrm{s}_1\, \mathrm{s}_2 \ldots \in \mathrm{S}^\omega$ such that $(\mathrm{s}_i, \mathrm{w}_i, \mathrm{s}_{i+1}) \in \Delta$ for all $i \geq 0$. A word is accepted by A if there exists a run $\mathrm{s}_0\, \mathrm{s}_1 \ldots$ such that $\mathrm{s}_i \in \mathrm{F}$ for infinitely many i. We denote by $\mathrm{L(A)}$ the language of the Büchi automaton, i.e., the set of words accepted by A. A language that is accepted by a Büchi automaton is called ω-regular.

We use Linear Temporal Logic (LTL) [24] over a set of atomic propositions AP to specify the desired behavior of a machine. An LTL formula may refer to atomic propositions, Boolean operators, and the temporal operators *next* X and *until* U. Formally, an LTL formula φ is defined inductively as $\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathsf{X}\,\varphi \mid \varphi\ \mathsf{U}\ \varphi$ with $p \in \mathrm{AP}$. The semantics of an LTL formula φ is given with respect to words $\mathrm{w} \in \Sigma_\mathrm{AP}^\omega$ using the satisfaction relation $\models$. As usual, we define it inductively over the structure of the formula as follows: (i) $\mathrm{w} \models p$ iff $p \in \mathrm{w}_0$, (ii) $\mathrm{w} \models \neg\varphi$ iff $\mathrm{w} \not\models \varphi$, (iii) $\mathrm{w} \models \varphi_1 \wedge \varphi_2$ iff $\mathrm{w} \models \varphi_1$ and $\mathrm{w} \models \varphi_2$, (iv) $\mathrm{w} \models \mathsf{X}\,\varphi$ iff $\mathrm{w}_{1..} \models \varphi$, and (v) $\mathrm{w} \models \varphi_1\ \mathsf{U}\ \varphi_2$ iff $\exists i \geq 0 : \mathrm{w}_{i..} \models \varphi_2$ and $\forall j, 0 \leq j < i : \mathrm{w}_{j..} \models \varphi_1$. The Boolean operators $\vee$, $\rightarrow$, and $\leftrightarrow$ are derived as usual. We use the common abbreviations for false, true, F, and G, i.e., $\mathsf{false} := p \wedge \neg p$, $\mathsf{true} := \neg\mathsf{false}$, $\mathsf{F}\,\varphi := \mathsf{true}\ \mathsf{U}\ \varphi$, and $\mathsf{G}\,\varphi := \neg\,\mathsf{F}\,\neg\varphi$. For instance, every word w with $p \in \mathrm{w}_i$ for some $i \geq 0$ satisfies $\mathsf{F}\,p$. Dually, every word with $p \notin \mathrm{w}_i$ for all $i \geq 0$ satisfies $\mathsf{G}\,\neg p$. The language of φ, denoted $\mathrm{L}(\varphi)$, is the set of words satisfying formula φ. For every LTL formula φ one can construct a Büchi automaton A such that $\mathrm{L(A)} = \mathrm{L}(\varphi)$ [22, 36].

We will use the following lemma in Section 4. It follows directly from the definition (i.e., from the fact that δ is a complete function).

Lemma 1 (Machine languages). *The language* $\mathrm{L(M)}$ *of any machine* $\mathrm{M} = (\mathrm{Q}, \Sigma_\mathrm{I}, \Sigma_\mathrm{O}, \mathrm{q}_0, \delta, \gamma)$ *is* I*-deterministic (input deterministic) and* I*-complete (input complete).*

Realizability and Synthesis Problem. The synthesis problem [6] asks to construct a system that satisfies a given formal specification. Given a language language L over the atomic propositions AP partitioned into input and output propositions, i.e., $\mathrm{AP} = I \cup O$, and a finite state machine M with input alphabet Σ_I and output alphabet Σ_O, we say that M *implements* (*realizes, or satisfies*) L, denoted by $\mathrm{M} \models L$, if $\mathrm{L(M)} \subseteq L$. We say language L is *realizable* if there exists a machine M that implements L. An LTL-formula φ is realizable if $\mathrm{L}(\varphi)$ is realizable.

Theorem 1 (Synthesis Algorithms [3, 25, 27]). *There exists a deterministic algorithm that checks whether a given LTL-formula (or an* ω*-regular language)* φ *is* realizable. *If* φ *is realizable, then the algorithm constructs* M.

```
1  typedef enum {RED, YELLOW, GREEN} traffic_light;
2  module Traffic (clock, sensor1, sensor2, light1, light2);
3    input clock, sensor1, sensor2;
4    output light1, light2;
5    traffic_light reg light1, light2;
6    initial begin
7      light1 = RED;
8      light2 = RED;
9    end
10   always @(posedge clock) begin
11     case (light1)
12       RED: if (sensor1) // Repair : if(sensor1 & !(light2 == RED & sensor2))
13                 light1 = YELLOW;
14       YELLOW: light1 = GREEN;
15       GREEN: light1 = RED;
16     endcase // case (light1)
17     case (light2)
18       RED: if (sensor2)
19                   light2 = YELLOW;
20       YELLOW: light2 = GREEN;
21       GREEN: light2 = RED;
22     endcase // case (light1)
23   end // always (@posedge clock)
24 endmodule // traffic
```

Fig. 1. Implementation of a traffic light system and a repair

3 Example

In this section we give a simple example to motivate our definitions and highlight the differences to previous approaches such as [18].

Example 1 (Traffic Light). Assume we want to develop a sensor-driven traffic light system for a crossing of two streets. For each street entering the crossing, the system has two sets of lights (called `light1` and `light2`) and two sensors (called `sensor1` and `sensor2`). By default both lights are red. If a sensor detects a car, then the corresponding lights should change from red to yellow to green and back to red. We are given the implementation shown in Figure 1 as starting point. It behaves as follows: for each red light, the system checks if the sensor is activated (Line 12 and 18). If yes, this light becomes yellow in the next step, followed by a green phase and a subsequent red phase. Assume we require that our implementation is safe, i.e., the two lights are never green at the same time. In LTL, this specification is written as $\varphi = \mathsf{G}(\texttt{light1} \neq \texttt{GREEN} \vee \texttt{light2} \neq \texttt{GREEN})$. The current implementation clearly does not satisfy this requirement: if both sensors detect a car initially, then the lights will simultaneously move from red to yellow and then to green, thus violating the specification.

Following the approach in [18] we introduce a non-deterministic choice into the program and then use a synthesis procedure to select among these options in order to satisfy the specification. For instance, we replace Line 12 (in Figure 1) by `if(?)`

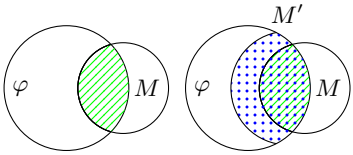


Fig. 2. Graphical representation of Def. 1

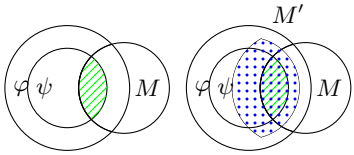


Fig. 3. Graphical representation of Def. 2

and ask the synthesizer to construct a new expression for `?` using the input and state variables. The synthesizer aims to find a simple expression s.t. φ is satisfied. In this case one simple admissible expression is `false`. It ensures that the modified program satisfies specification φ. While this repair is correct, it is very unlikely to please the programmer because it repairs "too much": it modifies the behavior of the system on input traces on which the initial implementation was correct. We believe it is more desirable to follow the idea of Chandra et al. [4] saying that a repair is only allowed to change the behavior of incorrect executions. In our case, the repair suggested above would not be allowed because it changes the behavior on correct traces, as we will show in the next section.

4 Repair

In this section we first give a repair definition for reactive systems which follows the intuition that a repair can only change the behavior of incorrect executions. Then, we provide an algorithm to compute such repairs.

4.1 Definitions

Given a machine M and a specification φ, we say a machine M' is an exact repair of M if (i) M' behaves like M on traces satisfying φ and (ii) if M' implements φ. Intuitively, the correct traces of M act as a *lower bound* for M' because they must be included in $\mathrm{L}(\mathrm{M}')$. $\mathrm{L}(\varphi)$ acts as an *upper bound* for M', i.e., it specifies the allowed traces.

Definition 1 (Exact Repair). *A machine M' is an exact repair of a machine M for a specification φ, if (i) all the correct traces of M are included in the language of M', and (ii) if the language of M' is included in the language of the specification φ, i.e.,*

$$\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi) \subseteq \mathrm{L}(\mathrm{M}') \subseteq \mathrm{L}(\varphi) \tag{1}$$

Note that the first inclusion defines the behavior of M' on all input words to which M responds correctly according to φ. In other terms, M' has only one choice forn inputs which M treat correctly. Figure 2 illustrates Definition 1: the two circles depict $\mathrm{L}(\mathrm{M})$ and $\mathrm{L}(\varphi)$. A repair has to (i) cover their intersection (first inclusion in Definition 1), which we depict with the striped area in the picture, and (ii) lie within $\mathrm{L}(\varphi)$ (second inclusion in Definition 1). One such repair is depicted by the dotted area on the right.

Example 2 (Traffic Light, cont.). The repair suggested in Example 1 (i.e., to replace `if (sensor1)` by `if (false)`) is not a valid repair according to Definition 1. The original implementation responds correctly, e.g., to the input trace in which `sensor1`

is always high and `sensor2` is always low, but the repair produces different outputs. The initial implementation behaves correctly on any input trace on which `sensor1` and `sensor2` are never high simultaneously. Any correct repair should include these input/output traces. An exact repair according to Definition 1 replaces `if (sensor1)` by `if (sensor1 & !(light2 == RED & sensor2))`. This repair retains all correct traces while avoiding the mutual exclusion problem.

While Definition 1 excludes the undesired repair in our example, it is sometimes too restrictive and can make repair impossible as the following example shows.

Example 3 (Definition 1 is too restrictive). Assume a machine M with input r and output g that always copies r to g, i.e., M satisfies $\mathsf{G}(r \leftrightarrow g)$. The specification requires that g is eventually high, i.e., $\varphi = \mathsf{F}\, g$. Definition 1 requires the repaired machine M' to behave like M on all traces on which M behaves correctly. M responds correctly to all input traces containing at least one r, i.e., $L(M) \cap L(\varphi) = \mathsf{F}(r \wedge g)$. Intuitively, M' has to mimic M as long as M still has a chance to satisfy φ (i.e., to produce a trace satisfying $\mathsf{F}(r \wedge g)$). Since M always has a chance to satisfy φ, M' has to behave like M in every step, therefore M' also violates φ, and cannot be repaired in this case.

In order to allow more repairs, we *relax* the restriction requiring that all correct traces are included in the following definition.

Definition 2 (Relaxed Repair). *Let ψ be a language (given by an LTL-formula or a Büchi automaton). We say M' is a* repair of M with respect to ψ and φ *if M' behaves like M on all traces satisfying ψ and M' implements φ. That is, M' is a repair constructed from M iff*

$$\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi) \subseteq \mathrm{L}(\mathrm{M}') \subseteq \mathrm{L}(\varphi) \tag{2}$$

In Figure 3 we give a graphical representation of this definition. The two concentric circles depict φ and ψ. (The definition does not require that $\mathrm{L}(\psi) \subseteq \mathrm{L}(\varphi)$, but for simplicity we depict it like that.) The overlapping circle on the right represents M. The intersection between ψ and M (the striped area in Figure 3) is the set of traces M' has to mimic. On the right of Figure 3, we show one possible repair (represented by the dotted area). The repair covers the intersection of $\mathrm{L}(\mathrm{M})$ and $\mathrm{L}(\psi)$, but not the intersection of $\mathrm{L}(\varphi)$ and $\mathrm{L}(\mathrm{M})$. The repair lies completely in $\mathrm{L}(\varphi)$. The choice of ψ influences the existence of a repair. In Section 5 we discuss several choices for ψ.

Example 4 (Example 3 continued). Example 3 shows that setting ψ to φ, i.e., $\mathsf{F}\, g$ in our example, can be too restrictive. If we relax ψ and require it only to include all traces in which g is true within the first n steps for some given n (i.e., $\psi = \bigvee_{i=0..n} \mathsf{X}^n\, g$), then we can find a repair. A possible repair is a machine M' that copies r to g in the first n steps and keeps track if g has been high within these steps. In this case, M' continues mimicing M, otherwise it set g to high in step $n+1$ independent of the behavior of M. This way M' satisfies the specification $(\mathsf{F}\, g)$ and mimics M for all traces satisfying ψ.

4.2 Reduction to Classical Synthesis

The following theorem shows that our repair problem can be reduced to the classical synthesis problem.

Theorem 2. *Let φ, ψ be two specifications and* M, M′ *be two machines with input signals I and output signal O. Machine* M′ *satisfies Formula 2* $(\mathrm{L(M)} \cap \mathrm{L}(\psi) \overset{(a)}{\subseteq} \mathrm{L(M')} \overset{(b)}{\subseteq} \mathrm{L}(\varphi))$ *if and only if* M′ *satisfies the following formula:*

$$\mathrm{L(M')} \subseteq \underbrace{\big((\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L(M)}\big)}_{(i)} \cap \underbrace{\mathrm{L}(\varphi)}_{(ii)} \tag{3}$$

For two languages A and B, $A \rightarrow B$ is an abbreviation for $(\Sigma^\omega \setminus A) \cup B$. Intuitively, Formula 3 requires that (i) M′ *behaves like* M *on all input words that* M *answers conforming to ψ and* (ii) M *satisfies specification φ.*

Proof. From left to right: We have to show that $\mathrm{L(M')}$ is included in (i) and (ii). Inclusion in (ii) follows trivially from (b). It remains to show $\mathrm{L(M')} \subseteq \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L(M)}$. Let $\mathrm{w} \in \mathrm{L(M')}$. If $\mathrm{w} \notin \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP}$, then the implication follows trivially. Otherwise we have to show that $\mathrm{w} \in \mathrm{L(M)}$. Since $\mathrm{w} \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP}$, it follows that $\mathrm{w}\downarrow_I \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I$. From $\mathrm{w}\downarrow_I \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I$ and the fact that $L(M)$ is input deterministic, we know that $M(\mathrm{w}\downarrow_I) \in \mathrm{L(M)} \cap \mathrm{L}(\psi) \subseteq \mathrm{L(M')}$ (due to (a)). Together with $\mathrm{L(M')}$ being input deterministic, it follows that $M(\mathrm{w}\downarrow_I) = M'(\mathrm{w}\downarrow_I) = \mathrm{w}$, and so $\mathrm{w} \in \mathrm{L(M)}$ holds.

From right to left: We have to show (a) and (b). (b) follows trivially from $\mathrm{L(M')} \subseteq (ii)$. It remains to show (a), i.e., that $\mathrm{L(M)} \cap \mathrm{L}(\psi) \subseteq \mathrm{L(M')}$. Assume a word $\mathrm{w} \in \mathrm{L(M)} \cap \mathrm{L}(\psi)$, we have to show that $\mathrm{w} \in \mathrm{L(M')}$. Let $\mathrm{w'} \in \mathrm{L(M')}$ be a word such that $\mathrm{w}\downarrow_I = \mathrm{w'}\downarrow_I$. Note that w′ exists because $\mathrm{L(M')}$ is input complete. We now show that $\mathrm{w} = \mathrm{w'}$, which implies that $\mathrm{w} \in \mathrm{L(M')}$. Since $\mathrm{w} \in \mathrm{L(M)} \cap \mathrm{L}(\psi)$, it follows that $\mathrm{w}\downarrow_I (= \mathrm{w'}\downarrow_I) \in (\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I$. Therefore, we know that $\mathrm{w'} \in \big(\mathrm{L(M)} \cap \mathrm{L}(\psi)\big)\downarrow_I\uparrow_{AP}$. From $\mathrm{L(M')} \subseteq (i)$ and from $\mathrm{w'} \in \mathrm{L(M')}$, it follows that $\mathrm{w'} \in \mathrm{L(M)}$. Since $\mathrm{L(M)}$ is input deterministic, $\mathrm{w} \in \mathrm{L(M)}$, $\mathrm{w'} \in \mathrm{L(M)}$, and $\mathrm{w}\downarrow_I = \mathrm{w'}\downarrow_I$, it follows that $\mathrm{w} = \mathrm{w'}$.

This theorem leads together with [25] to the following corollary, which allows us to use classical synthesis algorithms to compute repairs.

Corollary 1 (Existence of repair). *A repair can be constructed from a machine* M *with respect to specifications ψ and φ if and only if the language*

$$\big((\mathrm{L(M)} \cap \mathrm{L}(\psi))\downarrow_I\uparrow_{AP} \rightarrow \mathrm{L(M)}\big) \cap \mathrm{L}(\varphi) \tag{4}$$

is realizable.

4.3 Algorithm

Corollary 1 gives an algorithm to construct repairs based on synthesis techniques (cf. [18]). In order to compute the language defined by Formula 4, we can use standard automata-theoretic operations. More precisely, we construct a Büchi automaton A_φ recognizing φ and a Büchi automaton A_ψ recognizing ψ. Note that M is a Büchi automaton in which all states are accepting. Since Büchi automata are closed under conjunction, disjunction, projection, and complementation, we can construct an automaton

for $\big(\overline{(\mathrm{M}\times\mathrm{A}_\psi)|_I}+\mathrm{M}\big)\times\mathrm{A}_\varphi$, where by $A\times B$ denotes the conjunction, $A+B$ denotes the disjunction of automata A and B, $\bar{A}$ denotes the complementation of A, and $A|_I$ the projection of automaton A with respect to a set of proposition I. Once we have a Büchi automaton for the language in Formula 4, we can use Theorem 1 to synthesize a repair.

This algorithm in unlikely to scale because the complementation of a Büchi automaton induces an exponential blow-up in the worst case [10]. Furthermore, the projection operator can introduce non-determinism that can complicate the application of a synthesis procedure due to the need of an additional determinization step, leading to another exponential blow-up [23, 32]. In the following we show how to obtain an efficient algorithm by avoiding complementation (Lemma 2) and projection (Lemma 3).

Lemma 2. *Given a machine* M *with input signals* I *and output signals* O *and an LTL-formula* φ *over the atomic propositions* $AP = I \cup O$*, the following equalities hold:*

$$\Sigma_I^\omega \setminus \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\varphi)\big)\downarrow_I = \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi)\big)\downarrow_I \tag{5}$$

$$\Sigma_{AP}^\omega \setminus \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\varphi)\big)\downarrow_I\uparrow_{AP} = \big(\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi)\big)\downarrow_I\uparrow_{AP} \tag{6}$$

Proof. Intuitively, Equation 5 means that the set of input words on which M behaves correctly, i.e., satisfies φ, is the complement of the set of inputs on which M behaves incorrectly, i.e., violates φ and therefore satisfies $\neg\varphi$. Formally, we know from the semantics of LTL that $\mathrm{L}(\neg\varphi) = \Sigma^\omega \setminus \mathrm{L}(\varphi)$, which implies that

$$\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\varphi) \overset{(a)}{=} \mathrm{L}(\mathrm{M})\cap\big(\Sigma^\omega\setminus\mathrm{L}(\varphi)\big) \overset{(b)}{=} \mathrm{L}(\mathrm{M})\setminus\mathrm{L}(\varphi). \tag{7}$$

Equality 7.b follows from simple set theory. Furthermore, since $\mathrm{L}(\mathrm{M})$ is input deterministic and input complete, we know that

$$\forall\, w, w' \in \mathrm{L}(\mathrm{M}) : (w\downarrow_I = w'\downarrow_I) \rightarrow w = w' \tag{8}$$

$$\forall\, w \in \Sigma_{AP}^\omega : \exists\, w' \in \mathrm{L}(\mathrm{M}) : w\downarrow_I = w'\downarrow_I \tag{9}$$

We use these facts to show that for all $A \subseteq \Sigma^\omega$, $\Sigma_I^\omega \setminus \big(\mathrm{L}(\mathrm{M})\cap A\big)\downarrow_I = \big(\mathrm{L}(\mathrm{M})\setminus A\big)\downarrow_I$ holds, which proves together with Equation 7 that Equation 5 is true.

$v \in \big(\mathrm{L}(\mathrm{M})\setminus A\big)\downarrow_I \iff \exists\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\setminus A : (\mathrm{w}\downarrow_I = v) \iff \exists\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : (\mathrm{w}\downarrow_I = v)\wedge \mathrm{w}\notin A \overset{\text{Eq.8}}{\underset{\text{Eq.9}}{\iff}} \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : (\mathrm{w}\downarrow_I = v)\rightarrow \mathrm{w}\notin A \iff \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M}) : \mathrm{w}\in A\rightarrow(\mathrm{w}\downarrow_I\neq v) \iff \forall\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\cap A : (\mathrm{w}\downarrow_I\neq v) \iff \nexists\,\mathrm{w}\in\mathrm{L}(\mathrm{M})\cap A : (\mathrm{w}\downarrow_I = v) \iff v\notin\big(\mathrm{L}(\mathrm{M})\cap A\big)\downarrow_I$

Equation 6 is a simple extension of Equation 5 to the alphabet Σ_{AP}. It follows from the fact that for any language $\mathrm{L}\subseteq\Sigma_I^\omega : (\Sigma_I^\omega\setminus\mathrm{L})\uparrow_{AP} = \Sigma_I^\omega\uparrow_{AP}\setminus\mathrm{L}\uparrow_{AP}$ holds.

With the help of Lemma 2 we can simplify Formula 4 to

$$\big((\mathrm{L}(\mathrm{M})\cap\mathrm{L}(\neg\psi))\downarrow_I\uparrow_{AP}\cup\mathrm{L}(\mathrm{M})\big)\cap\mathrm{L}(\varphi) \tag{10}$$

This allows us to compute a repair using a synthesis procedure for the automaton $\big((\mathrm{M}\times\mathrm{A}_{\neg\psi})|_I+\mathrm{M}\big)\times\mathrm{A}_\varphi$, which is much simpler to construct.

Lemma 3 (Avoiding input projection). *Given a machine* M *and an LTL-formula* φ*, for every word* $\mathrm{w} \in \Sigma^\omega$*,* $\mathrm{w} \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP} \iff \mathrm{M}(\mathrm{w}\downarrow_I) \in \mathrm{L}(\varphi)$ *holds.*

Proof. $\mathrm{w} \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP} \iff \mathrm{w}\downarrow_I \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I \iff \exists\, \mathrm{w}' \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi) : \mathrm{w}'\downarrow_I = \mathrm{w}\downarrow_I \iff \exists\, \mathrm{w}' \in \mathrm{L}(\mathrm{M}) : \mathrm{w}'\downarrow_I = \mathrm{w}\downarrow_I \wedge \mathrm{w}' \in \mathrm{L}(\varphi) \iff \mathrm{M}(\mathrm{w}\downarrow_I) \in \mathrm{L}(\varphi)$

Due to Lemma 3 we can check if a word produced by M' lies in $(\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\varphi))\downarrow_I\uparrow_{AP}$ by checking whether M treats the input projection of that word correctly. A synthesizer looking for a solution to Equation 10 can simulate M and check its output against $\neg\psi$ to decide whether M' is allowed to deviate from M. This allows us to solve our repair problem using the simple setup depict in Figure 4. It shows five automata running in parallel:

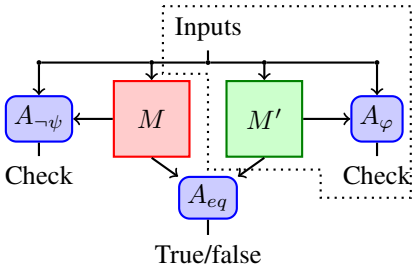


Fig. 4. Efficient implementation

1. The original machine M.
2. The repair candidate M', a copy of M that includes multiple options to modify M.
3. A specification automaton A_φ to check if the new machine M' satisfies its objective.
4. A specification automaton $A_{\neg\psi}$ to check if the original machine M violates ψ.
5. A specification automaton A_{eq} that checks if the outputs of M and M' coincide, i.e., $eq = \mathsf{G}(\bigwedge_{o\in O} o \leftrightarrow o')$, where O is the set of outputs of M and o' is the copy of output $o \in O$ in machine M'.

Theorem 3. *Given the setup depicted in Figure 4, a repair option in* M' *is a valid repair according to Definition 2, if it satisfies the formula*

$$\varphi \wedge (\neg\psi \vee eq). \tag{11}$$

Proof. Follows from Lemma 2 and Lemma 3.

Formula 11 forces M' to (1) behave according to φ and (2) mimic the behavior of M, if M satisfies ψ. Note that all automata can be constructed separately because they can be connected through the winning (or acceptance) condition. We avoid the monolithic construction of a specification automaton and obtain the same complexity as for classical repair. E.g., if φ, $\neg\psi$, and eq are represented by Büchi automata, then we can check for $\varphi \wedge (\neg\psi \vee eq)$ by first merging the acceptance states of $\neg\psi$ and eq, and then solving for a generalized Büchi condition, which is quadratic in the size of the state space $(|A_{\neg\psi}| \times |M| \times |M'| \times |A_\varphi| \times 2)$.

4.4 Implementation

Our prototype implementation is based on the following two ideas:

1. If a synthesis problem can be decided by looking at a finite set of possible repairs[1] (combinations of choices), then the choice of repair can be encoded using multiple initial states.

[1] Note that any synthesis problem with memoryless winning strategies satisfies this condition.

2. An initial state that does not lead to a counter example represents a correct repair. Any model checker can be adapted to return such an initial state, if one exists. By default a model checker returns the opposite, i.e., an initial state that leads to a counter-example but it is not difficult to change it. E.g., in BDD-based model-checkers some simple set operations suffice and in SAT-based checkers one can make use of unsat-core to eliminate failing initial states.

The main drawback of this approach is that the state space is multiplied by the number of considered repairs. However, the approach has several benefits which make it particularly interesting for program repair. First, it is easy to restrict the set of repairs to those that are simple and readable. In our prototype implementation we adapt the idea of Solar-Lezama et al. [33] and search for a repair within a given set of user-defined expressions. In the examples, we derive these expressions manually from the operators used in the program (see Section 6 for more details). Furthermore, we assume a given fault location that will be replaced by one of the user-defined expressions (cf. [18, 19]). Expression generation and fault localization are interesting and active research directions (cf. Section 1) but are not addressed in this paper. We focus on the problem of deciding what constitutes a good repair. The second main benefit is that we can adapt an arbitrary model checker to solve our repair problem. We believe (based on initial experiments) that at the current state, model checkers are significantly more mature than synthesis frameworks. In our implementation we used a version of NuSMV [7] that we slightly modified to return an initial state that does not lead to a counter example.

5 Discussion and Limitations

In this section we discuss choices for ψ and analyze why a repair can fail.

5.1 Choices for ψ

We present three different choices for ψ and analyze their strengths and weaknesse:
(1) $\psi = \varphi$, (2) if $\varphi = f \rightarrow g$, then $\psi = f \wedge g$, and (3) $\psi = \emptyset$.

Exact. Choosing $\psi = \varphi$ is the most restrictive choice. It requires that M' behaves like M on all words that are correct in M. While this is in general desirable, this choice can be too restrictive as Example 3 in Section 4 shows. One might think that the problem in Example 3 is that φ is a liveness specification. The following example shows that choosing $\psi = \varphi$ can also be too restrictive for safety specifications.

Example 5. Let M be a machine with input r and output g; M always outputs $\neg g$, i.e., M implements $\mathsf{G}(\neg g)$. Assume $\varphi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g) = \mathsf{G}(r) \vee \mathsf{G}(g)$. Applying Formula 10, we obtain $(\mathsf{G}(\neg g) \wedge \neg(\mathsf{G}(r) \vee \mathsf{G}(g)))\downarrow_I\uparrow_{AP}{}^2 \wedge (\mathsf{G}(r) \vee \mathsf{G}(g)) = (\mathsf{F}(\neg r) \wedge \mathsf{G}(g)) \vee (\mathsf{G}(r) \wedge \mathsf{G}(\neg g))$. This formula is not realizable because a machine does not know if the environment will always send a request ($\mathsf{G}(r)$) or if the environment will eventually stop sending a request ($\mathsf{F}(\neg r)$). A correct machine has to respond differently in these two cases. So, M cannot be repaired if $\psi = \varphi$.

[2] LTL is not closed under projection. We use LTL only to describe the corresponding automata computations.

Assume-Guarantee. It is very common that the specification is of the form $f \rightarrow g$ (as in the previous example). Usually, f is an assumption on the environment and g is the guarantee the machine has to satisfy if the environment meets the assumption. Since we are only interested in the behavior of M if the assumption is satisfied, it is reasonable to ask the repair to mimic only traces on which the assumption and the guarantee is satisfied, i.e., choosing $\psi = f \wedge g$.

Example 6 (Example 5 continued). Recall Example 5, we decompose φ into assumption $\mathsf{F}\,\neg r$ and guarantee $\mathsf{G}\,g$. Now, we can see that M is only correct on words on which the assumption is violated, so the repair should not be required to mimic the behavior of M. If we set $\psi = \mathsf{F}\,\neg r \wedge \mathsf{G}\,g$, then $L(M) \cap L(\psi) = \emptyset$ and M' is unrestricted on all input traces.

Unrestricted. If we choose $\psi = \emptyset$ the repair is unrestricted and the approach coincides with the work presented in [18].

5.2 Reasons for Repair Failure

In the following we discuss why a repair attempt can fail. The first and simplest reason is that the specification is not realizable. In this case, there is no correct system implementing the specification and therefore also no repair. However, a machine can be unrepairable even with respect to a realizable specification. The existence of a repair is closely related to the question of realizability (Corollary 1). Rosner [30] identified two reasons for a specification φ to be unrealizable.

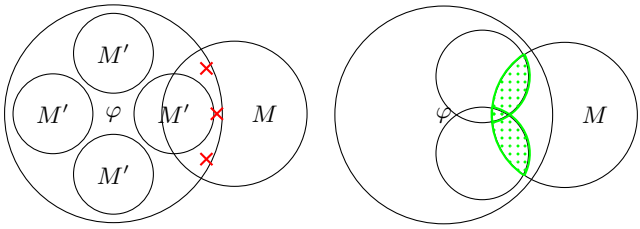
1. **Input-Completeness:** if φ is not input-complete, then φ is not realizable. For instance, consider specification $\mathsf{G}(r)$ requiring that r is always true. If r is an input to the system, the system cannot choose the value of r and therefore also not guarantee satisfaction of φ.
2. **Causality/Clairvoyance:** certain input-complete specifications can only be implemented by a clairvoyant system, i.e., a system that has knowledge about future inputs (a system that is non-causal). For instance, if the specification requires that the current output is equal to the next input, written as $\mathsf{G}(o \leftrightarrow \mathsf{X}\,i)$, then a correct system needs a look-ahead of size one to produce a correct output.

The following lemma shows that given an input-complete specification φ, input-completeness will not cause our repair algorithm to fail.

Lemma 4 (Input-Completeness). *If φ is input-complete, then $\big((\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))\downarrow_I \rightarrow \mathrm{L}(\mathrm{M})\big) \cap \mathrm{L}(\varphi)$ is input-complete.*

Proof. Let $\mathrm{w}_I \in {\Sigma_\mathrm{I}}^\omega$. If $\mathrm{w}_I \in (\mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi))\downarrow_I$, then there is a word $\mathrm{w} \in \mathrm{L}(\mathrm{M}) \cap \mathrm{L}(\psi)$ such that $\mathrm{w}\downarrow_I = \mathrm{w}_I$. Therefore we have found a word for w_I. If not, then a word for w_I exists because φ is input complete.

A failure due to missing causality can be split into two cases: the case in which the repair needs finite look-ahead (see Example 7 below) and the case in which it needs infinite look-ahead (see Example 8 below). The examples show that even if the specification is realizable (meaning implementable by a causal system), the repair might not be implementable by a causal system.



(a) M includes bad traces (b) M cuts two valid machines

Fig. 5. Two reasons for unrepairability

Example 7. Consider the realizable specification $\varphi = g \vee \mathsf{X}\, r$ and a machine M that keeps g low all the time, i.e., M satisfies $\mathsf{G}(\neg g)$. If input r is high in the second step, M satisfies φ. An exact repair (according to Definition 1) needs to set g to low in the first step if the input in the second step is $high$, because it has to mimic M in this case. On the other hand, it the input in the second step is low, g needs to be set to high in the first step. So, any exact repair has to have a look-ahead of at least one, in order to react correctly.

The following example shows a faulty machine and a (realizable) specification for which a correct repair needs infinite look-ahead.

Example 8. Consider a machine M with input r and output g that copies the input to the output. Assume we search for a repair such that the modified machine satisfies the specification $\varphi = \mathsf{G}\,\mathsf{F}\, g$ requiring that g is high infinitely often. Machine M violates the specification on all input sequences that keep r low from some point onwards, i.e., on all words fulfilling $\mathsf{F}(\mathsf{G}\, r)$. Recall that a repair M' has to behave like M on all correct inputs. In this example, M' has to behave like M on all finite inputs, because it does not know whether or not the input word lies in $\mathsf{F}(\mathsf{G}\, r)$ without seeing the word completely, i.e., without infinite look-ahead.

Theorem 4 (Possibility of repair). *Assume that we cannot repair machine* M *with respect to a realizable specification* φ*. Then, a repairing machine needs either finite or infinite look-ahead.*

Proof. Follows from [30], Corollary 1, and Lemma 4.

Characterization Based on Possible Machines. Another way to look at a failed repair attempt is from the perspective of possible machines. Recall, in Figure 3 we depict a correct repair M' as a circle covering the set of words in the intersection of M and ψ. In Figure 5 we use the same graphical representations to explain two reasons for failure. Figure 5(a) depicts several machines M' realizing φ. A repair of M has to be one of the machines realizing φ. As observed in [13], there are words satisfying φ that cannot be produced by any correct machine (depicted as red crosses in Figure 5(a)). E.g, recall the specification $\varphi = g \vee \mathsf{X}(r)$ in Example 7. The word in which g is low initially and r high in the second step satisfies φ but will not be produced by any correct (causal)

machine because the machine cannot rely on the environment to raise r in the second step. If the machine we are aiming to repair includes such a trace, a repair attempt with $\psi = \varphi$ will fail. In this case, we can replace φ (or ψ) by the strongest formula that is open-equivalent[3] to φ in order to obtain a solvable repair problem. However, even if φ is replaced by its strongest open-equivalent formula, the repair attempt might fail for the reason depicted in Figure 5(b). We again depict several machines M' realizing φ. M shares traces with several of these machines, but no machine covers the whole intersection of φ and M. In other words, an implementing machine would have to share the characteristics of two machines.

6 Empirical Results

In this section we first describe the repair we synthesized for the traffic light example from Section 3. Then, we summarize the results on a set of example we analyzed. All experiments were run on a 2.4GHz Intel(R) Core(TM)2 Duo laptop with 4 GB of RAM.

Traffic Light Example. In the traffic light example, we gave the synthesizer the option to choose from 2^{50} expressions (all possible logical expression over combinations of light colors and signal states). NuSMV returns the expression $(s_2 \wedge s_1 \wedge (l_2 \neq \text{RED})) \vee (\neg s_2 \wedge s_1 \wedge l_2 \neq \text{GREEN})$, which is equivalent to $s_1 \wedge ((s_2 \wedge l_2 \neq \text{RED}) \vee (\neg s_2 \wedge l_2 \neq \text{GREEN}))$ in 0.2 seconds. The repair forbids the first light from turning yellow if the second light is already green. This is not the repair we suggested in Section 3 because the synthesizer has freedom to choose between the expressions that satisfy the new notion. Our new approach avoids the obvious but undesired repair of leaving the first light red, irrespective of an arriving car. This is the solution NuSMV provides (within 0.16s) if we use the previous repair notion [18].

Experiments. In order to empirically test the viability of our approach and to confirm our improved repair suggestions, we applied our approach to several examples. We report the results in Table 1; For each example, we report the number of choices for the synthesizer (Column #Repairs), the time and number of BDD variables to (1) verify the correctness of the repair that we obtain (Column Verification), (2) find a repair with our new approach (Column Repair), and (3) solve the classical repair problem (Column Classical Repair).

In order to synthesize a repair, we followed the approach described in Section 4.4 (Figure 4), i.e., we manually added freedom to the model and wrote formula for $\neg\psi$ and equality checking. The examples are described in detail in the extended version[4] For all but one of the examples (Processor (1)), the previous approach synthesizes degenerated repairs, while our approach leads to a correct program repair.

Assume-Guarantee ($\rightarrow$) is Example 5 from Section 5.1. It uses the original specification for ψ, i.e., $\psi = \mathsf{F}(\neg r) \rightarrow \mathsf{G}(g)$. We let the synthesizer choose between all possible boolean combinations of g, r and a memory bit containing the previous value of g. Our approach fails to find a repair. Assume-Guarantee ($\&$) is Example 6 from Section 5.1

[3] Two formulas φ and φ' are open-equivalent if any machine M implementing φ also implements φ' and vice-versa [13].

[4] The NuSMV models of the example and our implementation are available at `http://www-verimag.imag.fr/~vonessen/`

with $\psi = \mathsf{F}(\neg r) \wedge \mathsf{G}(g)$, using the same potential repairs. In this case, a valid repair is found. The Binary Search examples model a binary search algorithm with a specification analogous to $sorted \rightarrow correct$, i.e., when the array is sorted, then the algorithm responds with the correct result. The bug is an incorrect assignment of the pointer into the array. The repais we allow are arithmetic combinations of the previous position, 1, -1, the lower bound and the upper bound. As in the Assume-Guarantee examples, we have two different choices for ψ here. In the case that $\psi = \varphi$, there is no repair available, while for $\psi = sorted \wedge correct$ we find the correct repair. The RW-Lock example demonstrates that our approach can also be used to synthesize locks. We require that only those program runs are changed that lead to a dead-lock, thereby synthesizing the minimum amount of locks. The potential repairs allow 16 different locking combinations, only one of which is optimal. The optimal solution is the only one admitted by our repair definition.

The Processor examples demonstrate what happens in complex models when increasing the amount of freedom in a model. They also show how repairing partial specifications may lead to the introduction of new bugs. In Processor (1), the minimal amount of non-determinism is introduced, i.e., only as much freedom as strictly necessary to repair. Here, the classical approach and our new approach give the same result. In Processor (2), we introduce more freedom, which leads to incorrect repairs with the classical approach. In particular, the fault is in the ALU of the processor, and the degenerated repairs incorrectly execute the `AND` instruction, which is handled correctly in the original model. We allow replacing the faulty and the a correct instruction by either a `XOR, AND, OR, SUB` or `ADD` instruction. Finally, Processor (3) shows that the time necessary for synthesis grows sub-linearly with the number of repair options.

On average, synthesizing a repair takes 2.3 times more time than checking its correctness. Our new approach seems to be one order of magnitude slower than the classical approach. This is expected because finding degenerated repairs is usually much simpler. (This is comparable to finding trivial counter examples.).In order to find correct repairs with the approach of [18], we would need to increase the size of the specification, which will significantly slow down the approach.

Table 1. Experimental results

		Verification		Repair		Classical Repair	
	#Repairs	time	#Vars	time	#Vars	time	#Vars
Assume-Guarantee ($\rightarrow$)	2^{12}	n/a	n/a	0.038	16	0.012	14
Assume-Guarantee (&)	2^{12}	0.015	14	0.025	14	0.012	12
Binary Search ($\rightarrow$)	5	n/a	n/a	0.78	27	0.1	21
Binary Search (&)	5	0.232	27	0.56	27	0.1	21
RW-Lock	16	0.222	34	0.232	34	0.228	22
Traffic	2^{55}	0.183	68	0.8	68	0.155	63
PCI	27	0.3	56	0.8	56	0.5	53
Processor (1)	2	2m02s	135	2m41s	135	0.5	69
Processor (2)	4	4m28s	138	5m07s	138	0.5	69
Processor (3)	25	5m23s	140	18m05s	140	0.5	71

7 Future Work and Conclusions

Future Work. We will follow two orthogonal directions to make it possible to repair more machines. The first one increases the computational power of a repaired machine. Every machine M' repairing M has to behave like M until it concludes that M does not respond to the remaining input word correctly. As shown in Example 7, M' might not know early enough if M will fail or succeed. Therefore, studying repairs with finite look-ahead is an interesting direction. The second direction studies a relaxed notion of set-inclusion or equality in order to express how "close" two machines are. To extend the applicability of our approach to infinite-state programs, we will explore suitable program abstraction techniques (cf. [34]). Finally, we are planing to experiment with model checkers specialized in solving the sequential equivalence checking problem [20, 21]. We believe that such solvers perform well on our problem, because M' and M have many similar structures.

Conclusion. When fixing programs, we usually fix bugs one by one; at the same time, we try to leave as many parts of the program unchanged as possible. In this paper, we introduced a new notion of program repair that supports this method. The approach allows an automatic program repair tool to focus on the task at hand instead of having to look at the entire specification. It also facilitates finding repairs for programs with incomplete specifications, as they often show up in real word programs.

References

1. Ball, T., Naik, M., Rajamani, S.K.: From symptom to cause: Localizing errors in counterexample traces. In: POPL 2003, pp. 97–105 (January 2003)
2. Buccafurri, F., Eiter, T., Gottlob, G., Leone, N.: Enhancing model checking in verification by ai techniques. Artif. Intell. 112(1-2), 57–104 (1999)
3. Büchi, J.R., Landweber, L.H.: Solving sequential conditions by finite-state strategies. Transactions of the American Mathematical Society 138, 295–311 (1969)
4. Chandra, S., Torlak, E., Barman, S., Bodik, R.: Angelic debugging. In: ICSE 2011, pp. 121–130. ACM, New York (2011)
5. Chang, K.H., Markov, I.L., Bertacco, V.: Fixing design errors with counterexamples and resynthesis. IEEE Trans. on CAD 27(1), 184–188 (2008)
6. Church, A.: Logic, arithmetic and automata. In: Proc. 1962 Int. Congr. Math. (1963)
7. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An OpenSource Tool for Symbolic Model Checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, p. 359. Springer, Heidelberg (2002)
8. Clarke, E., Grumberg, O., McMillan, K., Zhao, X.: Efficient generation of counterexamples and witnesses in symbolic model checking. In: DAC (1995)
9. Clarke, E.M., Emerson, E.A.: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. In: Grumberg, O., Veith, H. (eds.) 25MC Festschrift. LNCS, vol. 5000, pp. 196–215. Springer, Heidelberg (2008)
10. Drusinsky, D., Harel, D.: On the power of bounded concurrency i: Finite automata. J. ACM 41(3), 517–539 (1994)
11. Ebnenasir, A., Kulkarni, S.S., Bonakdarpour, B.: Revising unity programs: Possibilities and limitations. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 275–290. Springer, Heidelberg (2006)

12. Edelkamp, S., Lluch-Lafuente, A., Leue, S.: Trail-directed model checking. ENTCS 5(3) (August 2001); Software Model Checking Workshop 2001
13. Greimel, K., Bloem, R., Jobstmann, B., Vardi, M.: Open implication. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 361–372. Springer, Heidelberg (2008)
14. Griesmayer, A., Bloem, R., Cook, B.: Repair of boolean programs with an application to c. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 358–371. Springer, Heidelberg (2006)
15. Groce, A., Visser, W.: What went wrong: Explaining counterexamples. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 121–135. Springer, Heidelberg (2003)
16. Janjua, M.U., Mycroft, A.: Automatic correction to safety violations in programs. In: Thread Verification (TV 2006) (2006) (unpublished)
17. Jin, H., Ravi, K., Somenzi, F.: Fate and free will in error traces. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 445–459. Springer, Heidelberg (2002)
18. Jobstmann, B., Griesmayer, A., Bloem, R.: Program repair as a game. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 226–238. Springer, Heidelberg (2005)
19. Jobstmann, B., Staber, S., Griesmayer, A., Bloem, R.: Finding and fixing faults. J. Comput. Syst. Sci. 78(2), 441–460 (2012)
20. Kaiss, D., Skaba, M., Hanna, Z., Khasidashvili, Z.: Industrial strength sat-based alignability algorithm for hardware equivalence verification. In: FMCAD, pp. 20–26 (2007)
21. Khasidashvili, Z., Moondanos, J., Kaiss, D., Hanna, Z.: An enhanced cut-points algorithm in formal equivalence verification. In: HLDVT, pp. 171–176 (2001)
22. Lichtenstein, O., Pnueli, A.: Checking that finite state concurrent programs satisfy their linear specification. In: POPL, pp. 97–107 (1985)
23. Piterman, N.: From nondeterministic buchi and streett automata to deterministic parity automata. Logical Methods in Computer Science 3(3), 5 (2007)
24. Pnueli, A.: The temporal logic of programs. In: FOCS. IEEE Comp. Soc. (1977)
25. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: POPL, pp. 179–190 (1989)
26. Queille, J.P., Sifakis, J.: Specification and verification of concurrent systems in CESAR. In: Symposium on Programming, pp. 337–351 (1982)
27. Rabin, M.O.: Decidability of second-order theories and automata on infinite trees. Transactions of the American Mathematical Society 141, 1–35 (1969)
28. Ravi, K., Somenzi, F.: Minimal assignments for bounded model checking. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 31–45. Springer, Heidelberg (2004)
29. Renieris, M., Reiss, S.P.: Fault localization with nearest neighbor queries. In: ICASE, Montreal, Canada, pp. 30–39 (October 2003)
30. Rosner, R.: Modular Synthesis of Reactive Systems. PhD thesis, Stanford University (1997)
31. Samanta, R., Deshmukh, J.V., Emerson, E.A.: Automatic generation of local repairs for boolean programs. In: Cimatti, A., Jones, R.B. (eds.) FMCAD, pp. 1–10 (2008)
32. Schewe, S.: Tighter bounds for the determinisation of büchi automata. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 167–181. Springer, Heidelberg (2009)
33. Solar-Lezama, A., Rabbah, R.M., Bodík, R., Ebcioglu, K.: Programming by sketching for bit-streaming programs. In: PLDI, pp. 281–294 (2005)
34. Vechev, M.T., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)
35. Vechev, M., Yahav, E., Yorsh, G.: Inferring synchronization under limited observability. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 139–154. Springer, Heidelberg (2009)
36. Wolper, P., Vardi, M.Y., Sistla, A.P.: Reasoning about infinite computation paths (extended abstract). In: FOCS, pp. 185–194. IEEE (1983)
37. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. IEEE Transactions on Software Engineering 28(2), 183–200 (2002)

Programs from Proofs – A PCC Alternative*

Daniel Wonisch, Alexander Schremmer, and Heike Wehrheim

University of Paderborn
Germany
{alexander.schremmer,wehrheim}@upb.de

Abstract. Proof-carrying code approaches aim at safe execution of untrusted code by having the code producer attach a safety proof to the code which the code consumer only has to validate. Depending on the type of safety property, proofs can however become quite large and their validation - though faster than their construction - still time consuming.

In this paper we introduce a new concept for safe execution of untrusted code. It keeps the idea of putting the time consuming part of proving on the side of the code producer, however, attaches no proofs to code anymore but instead uses the proof to *transform* the program into an *equivalent* but *more efficiently verifiable* program. Code consumers thus still do proving themselves, however, on a computationally inexpensive level only. Experimental results show that the proof effort can be reduced by several orders of magnitude, both with respect to time and space.

1 Introduction

Proof-Carrying Code (PCC) has been invented in the nineties of the last century by Necula and Lee [21,22]. It aims at the safe execution of untrusted code: (untrusted) code producers write code, ship it via (untrusted) mediums to code consumers (e.g., mobile devices) and need a way of ensuring the safety of their code, and in particular a way of convincing the consumers of it. To this end, code producers attach safety proofs to their code, and consumers validate these proofs. This approach is *tamper-proof*: malicious modifications of the code as well as of the proof get detected.

This general framework has been instantiated with a number of techniques, differing in the type of safety property considered, proof method employed and proof validation concept applied (e.g. [10,3,13]). However, there are also a number of disadvantages associated with PCC which have hindered its widespread application. First of all, we need a specific PCC instance for the type of safety properties our consumer is interested in. Looking at the large spectrum of properties covered, this might turn out to be a solvable problem. However, the instance that we find might only produce very large proofs (which need to be attached to the code) or might only have expensive (wrt. time and space) proof checking techniques.

* This work was partially supported by the German Research Foundation (DFG) within the Collaborative Research Centre "On-The-Fly Computing" (SFB 901).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 912–927, 2013.

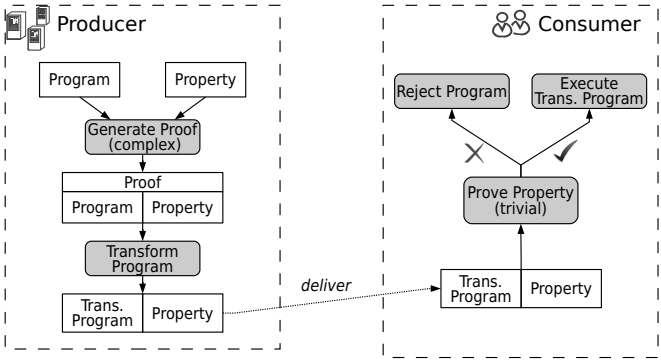


Fig. 1. Overview of our approach

In this paper we propose a different concept for the safe execution of untrusted code. Like PCC it is first of all a general concept with lots of possible instantiations, one of which will be introduced here. Our concept keeps the general idea behind PCC: the potentially untrusted code producer gets the major burden in the task of ensuring safety while the consumer has to execute a time and space efficient procedure only. Figure 1 gives a general overview of our approach: the producer carries out a proof of correctness of the program with respect to a safety property. The information gathered in the proof is next used to *transform* the program into an *equivalent*, more efficiently verifiable (but usually larger wrt. lines of code) program. The transformed program is delivered to the consumer who – prior to execution – is also proving correctness of the program, however, with a significantly reduced effort. The approach remains tamper-proof since the consumer is actually verifying correctness of the delivered program.

Key to the applicability of this concept is of course the possibility of finding instances of proof techniques, property classes and transformations actually exhibiting these characteristics. We are confident that a number of such instances can be found, and present one such instance as a proof of concept here. Our instantiation uses *software model checking* as proof technique on the producer side and *data flow analysis* on the consumer side.

More detailedly, the instance we present here allows for safety-property definitions in terms of *protocol* (or trace) specifications on function calls (or more general, program operations). The producer then employs a *predicate analysis* [16,1,4] on the program to prove properties. The predicate analysis constructs an abstract reachability tree (ART) giving us information about the possible execution paths of the program. The transformation uses this information to construct an equivalent program which has a control-flow graph (CFG) isomorphic to the ART. Original and transformed program are not only equivalent with respect to the behaviour but also with respect to performance (optimisations on the transformed program might even give us a program with less runtime). On the CFG of the transformed program the consumer can easily validate the absence

of errors using a simple data flow analysis which is linear in the size of the CFG (no iterative fixpoint computation needed).

In order to evaluate our approach, we have implemented it using the software verification tool CPAchecker [8] for predicate analysis and the tool Eli [17] to implement the simple data flow analysis. Evaluation on some standard benchmark C programs shows that the proof effort for the consumer is significantly lower than for the producer, even though the transformed program is usually larger (in terms of lines of code) than the original program.

2 Preliminaries

For the presentation in this paper, we consider programs to be written in a simple imperative single-threaded programming language with assignments, assume operations and function calls as only possible statements, and variables that range over integers only. The programs considered in the experimental results (Section 5) are however not restricted this way. The left of Figure 2 shows our running example of a program which is calling some lock and unlock functions. The objective is to show that this program adheres to common locking idioms (which it does), i.e., in particular no unlock can occur before a lock.

Formally, a *program* $P = (A, l_0)$ is represented as *control-flow automaton* (CFA) A together with a start location l_0. A CFA $A = (L, G)$ consists of a set of (program) locations L and a set of edges $G \subseteq L \times Ops \times L$ that describe possible transitions from one program location to another by executing certain operations Ops. A *concrete data state* $c : X \to \mathbb{Z}$ of a program P is a mapping from the set of variables X of the program to integer values. The set of all concrete data states in a program P is denoted by $\mathscr{C}$. A set of concrete data states can be described by a first-order predicate logic formula φ over the program variables (which we make use of during predicate abstraction). We write $[\![\varphi]\!] := \{c \in \mathscr{C} \mid c \models \varphi\}$ for the set of concrete data states represented by some formula φ. Furthermore, we write $\gamma(c)$ for the representation of a concrete data state as formula (i.e. $[\![\gamma(c)]\!] = \{c\}$). Note that we assume the program to be started in some arbitrary data state c_0.

A tuple (l, c) of a location and a concrete data state of a program is called *concrete state*. The *concrete semantics* of an operation $op \in Ops$ is defined in terms of the *strongest postcondition operator* $SP_{op}(\cdot)$. Intuitively, the strongest postcondition operator $SP_{op}(\varphi)$ of a formula φ wrt. to an operation op is the strongest formula ψ which represents all states which can be reached by op from a state satisfying φ. Formally, we have $SP_{x:=expr}(\varphi) = \exists \widehat{x} : \varphi_{[x \mapsto \widehat{x}]} \wedge (x = expr_{[x \mapsto \widehat{x}]})$ for an assignment operation $x := expr$, $SP_{assume(p)}(\varphi) = \varphi \wedge p$ for an assume operation $assume(p)$ ($assume(\cdot)$ omitted in figures) and $SP_{f()}(\varphi) = \varphi$ for a function call $f()$. Thus, we assume function calls to not change the data state of our program. Our implementation lifts this limitation, which was done only for presentation purposes.

```
1:  init();                          1:  init();
2:  lock();                          2:  lock();
3:  int lastLock = 0;                3:  int lastLock = 0;
4:  for(int i = 1; i < n; i++) {     4:  for(int i = 1; i < n; i++) {
5:    if( i - lastLock == 2 ) {      5:    unlock();
6:      lock();                      6:    i++;
7:      lastLock = i;                7:    if( i >=n ) { break; }
8:    } else {                       8:    lock();
9:      unlock();                    9:    lastLock = i;
10:    }                             10: }
11: }
```

Fig. 2. Program Locks (left) with its transformation to an equivalent program (right)

We write $(l, c) \xrightarrow{g} (l', c')$ for concrete states (l, c), (l', c') and edge $g := (l, op, l')$, if $c' \in [\![SP_{op}(\gamma(c))]\!]$. We write $(l, c) \to (l', c')$ if there is an edge $g = (l, op, l')$ such that $(l, c) \xrightarrow{g} (l', c')$. The feasible *paths* of a program $P = (A, l_0)$ with CFA $A = (L, G)$ are the sequence of concrete states the program can pass through:

$$paths(P) := \{(c_0, \ldots, c_n) \mid \exists l_1, \ldots, l_n \in L, \exists g_1, \ldots, g_{n-1} \in G : (l_0, c_0) \xrightarrow{g_1} \ldots \xrightarrow{g_{n-1}} (l_n, c_n)\}$$

Similarly, the set of *traces* of a program P are the sequences of operations it can perform:

$$traces(P) := \{(op_0 \ldots op_n \mid \exists l_1, \ldots, l_n \in L, \exists c_0, \ldots, c_n \in \mathscr{C}, \exists g_1, \ldots, g_{n-1} \in G : (l_0, c_0) \xrightarrow{g_1} \ldots \xrightarrow{g_{n-1}} (l_n, c_n) \wedge \forall 0 \leq i < n : g_i = (l_i, op_i, l_{i+1})\}$$

We are ultimately interested in proving *safety* properties of programs. Safety properties are given in terms of protocol automata which describe the allowed sequences of operations.

Definition 1. *A* protocol or property automaton $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$ *consists of an alphabet* Σ*, a finite set of states* S *with initial state* s_0 *and error state* s_{err}*, and transition relation* $\delta \subseteq S \times \Sigma \times S$*. The transition relation is deterministic. The error state has outgoing transitions* $(s_{err}, op, s_{err}) \in \delta$ *for all* $op \in \Sigma$*.*

The language $L(A_{prop})$ *of a protocol automaton is the set of traces* $op_1 \ldots op_n$ *such that* $\delta^*(s_0, op_1 \ldots op_n) \neq s_{err}$*.*

The property automaton in Figure 3 describes all valid locking patterns: first, a call of *init*() needs to be performed and then *lock*() and *unlock*() have to occur in turns. The operations occuring in property automata are usually function calls, however, these can also be any syntactic program property that is expressible as a BLAST automaton [5].

The property automaton only speaks about a part of the program operations, namely those in Σ. A comparison of program and protocol automaton traces thus

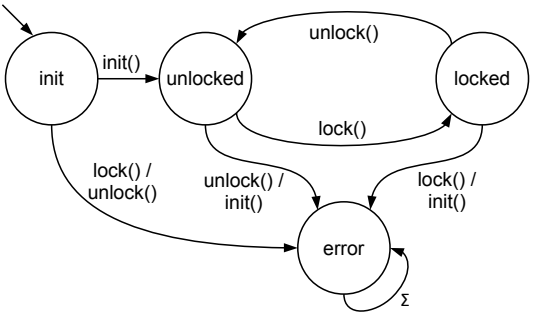


Fig. 3. Property Automaton. Disallows two lock() or unlock() in a row.

needs to project the traces of the program onto the alphabet of the automaton (projection written as $\restriction$). Hence, program P *satisfies* the safety property of protocol automaton A_{prop}, $P \models A_{prop}$, if $traces(P) \restriction \Sigma \subseteq L(A_{prop})$.

To analyse whether a safety property holds for a program, we use predicate abstraction (as supplied by CPAchecker [8]). CPAchecker builds the product of the property automaton and an abstraction of the concrete state space yielding an *abstract reachability tree* (ART). More precisely, the concrete data states are abstracted by quantifier-free first order predicate logic formulas over the variables of the program. We let PS denote the set of all such formulas. We only sketch the algorithm behind ART construction, for details see for instance [8]. What is important for us, is its form. The nodes of the ART take the form (l, s, φ) describing the location the program is in, the current state of the property automaton and a predicate formula as an abstraction of the data state. We assume that the ART is finite. This is the case if predicate refinement terminates for a given program.

Definition 2. *An* abstract reachability tree $T = (N, G, C)$ *consists of a set of nodes* $N \subseteq L \times S \times PS$*, a set of edges* $G \subseteq N \times Ops \times N$ *and a* covering $C : N \to N$*.*

The covering is used to stop exploration of the abstract state space at a particular node once we find that the node is covered by an already existing one: if $C(l, s, \varphi) = (l', s', \varphi')$ then $l = l'$, $s = s'$ and $[\![\varphi]\!] \subseteq [\![\varphi']\!]$. The successor of an already constructed node n is constructed by searching for successor nodes in the CFA, computing the abstract post operation on the predicate formula of n and determining the successor property automaton state. After generation of a new ART node, the algorithm checks whether the new ART node is covered by an existing one and generates an entry in the covering if necessary. One result of this process is that loops are unrolled such that within ART nodes program locations are only associated to a *single* state of the property automaton: if a program when reaching location l can potentially be in more than one state of the (concurrently running) protocol automaton, these will become separate nodes in the ART.

Figure 4 shows the ART of program LOCKS as constructed by CPAchecker. The dotted line depicts the covering. The ART furthermore satisfies some healthiness conditions which we need further on for the correctness of our construction (and which is guaranteed by the tool we use for ART construction):

Soundness. If $((l, s, \varphi), op, (l', s', \varphi')) \in G_{art}$, then for all $c \in [\![\varphi]\!]$ with $(l, c) \overset{(l,op,l')}{\rightarrow} (l', c')$ we have $c' \in [\![\varphi']\!]$. Furthermore, if $op \in \Sigma$, then $(s, op, s') \in \delta_{A_{prop}}$, and $s = s'$ else.

Completeness. If (l, op, l') is an edge in the CFG of the program, then for all $(l, s, \varphi) \in N \setminus \mathrm{dom}(C)$ with $\varphi \not\equiv false$, we have some $(l', \cdot, \cdot) \in N$ such that $((l, s, \varphi), op, (l', \cdot, \cdot)) \in G_{art}$.

Determinism. For every $n \in N$, there is only one successor node in G_{art} except when nodes have outgoing assume edges. In this case, also more than one successor nodes of n are allowed.

Well-Constructedness. For every $((l, \cdot, \cdot), op, (l', \cdot, \cdot)) \in G_{art}$, we have an edge (l, op, l') in the CFG. For $P = (A, l_0)$ and root of the ART (l_0, s_0, φ_0), we have $l = l_0$, $s = s_0$ and $[\![\varphi_0]\!] = \mathscr{C}$.

As a consequence, the program satisfies the property of the protocol automaton if the ART does not contain a node (l, s, φ) with $s = s_{err}$. Note that it is not always possible to construct a finite ART. In general, ART construction consists of incremental abstraction, checking and refinement steps (CEGAR) until the property is proven. However, our interest here is not in techniques for the initial proof of correctness by the producer of a program, but in a method for simplifying subsequent proofs by consumers. Thus we assume in the following that the program satisfies the property, and that we have an ART with the above mentioned characteristics available.

3 Program Transformation

The construction of the abstract reachability tree is the task of the producer. He or she needs to show that the program adheres to the safety property and needs a way of convincing the producer of this fact. Proving can be time-consuming, and this is an accepted property of PCC techniques. Proof checking on the consumer side should however be easy.

The first step of our method is now the transformation of the program into another program for which *proving* is much easier. So instead of "easy proof checking", in our approach the consumer is carrying out "easy proving".

The transformation proceeds by constructing – in its most basic form – a goto program from the ART. Later optimisations bring this into a more readable form, but here we just formally define transformation into goto programs by giving the new program again in the form of a control flow automaton and an initial location. The idea is quite simple: every node in the ART becomes a location in the new program, and the operations executed when going from one node to the next are those on the edges in the ART.

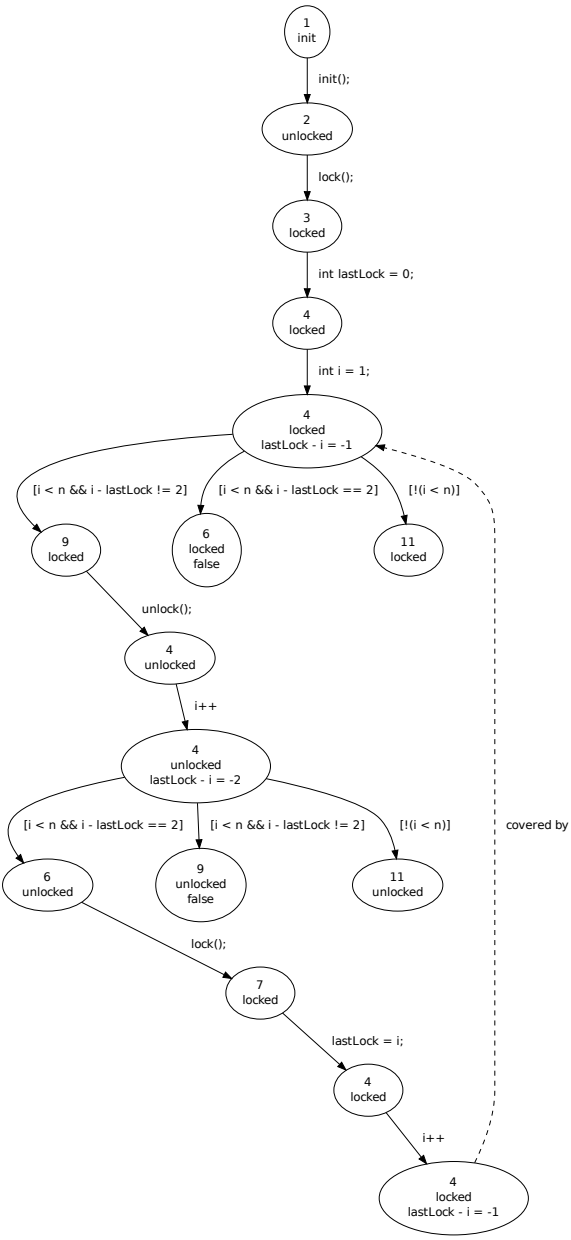


Fig. 4. Abstract reachability tree as generated by CPAchecker. Not all states bear abstraction predicates as the predicate analysis performs the abstraction only at loop heads in adjustable-block encoding mode [4].

Definition 3. *Let $T = (N, G, C)$ be an abstract reachability tree generated from a program and a property automaton. The* transformed program, *$program\_of(T)$, is a program $P' = ((L', G'), l'_0)$ with $L' = N \backslash dom(C)$, $l'_0 = root(T) = (l_0, s_0, true)$ and edges defined as*

$$(l_1, op, l_2) \in G' \Leftrightarrow \begin{cases} (l_1, op, l_2) \in G & \text{if } l_2 \notin dom(C) \\ (l_1, op, l_3) \in G \wedge (l_3, l_2) \in C & \text{otherwise.} \end{cases}$$

This is well defined because $\mathrm{dom}(G) \cap \mathrm{dom}(C) = \emptyset$, $C(N) \cap dom(C) = \emptyset$, and since the ART is deterministic. This representation can be easily brought back into a programming languge notation using gotos, and with some effort into a program without gotos and proper loops instead (assuming the resulting loop structure is reducible). The right of Figure 2 shows the transformed version of program LOCKS in the form which uses loops. Note that the transformation of a program containing gotos to the version containing loops is solely done for presentation purposes.

Due to the fact that the edges in the ART and thus the statements in the new program are exactly those of the original program, we obtain a new program which is equivalent to the original one: it possesses the same paths.

Theorem 1. *Let P be a program. Let $ART = (N, G_{art}, C)$ be an abstract reachablity tree for P. Let $P' = program\_of(ART)$. Then $paths(P) = paths(P')$.*

Furthermore, we perform a small optimisation that does not affect the correctness of the transformation. We also omit all edges in the ART leading to nodes labelled false as these represent steps the program will never execute. Furthermore, if after this removal there is just one outgoing edge from a node and this edge is labelled with an assume operation, we delete this assume.

The obtained program has exactly the same behaviour as the original program and thus the desired functionality. The lines of code usually increase during transformation. The performance (run time) however stays the same or even decreases. This is the case when the optimisation removes assume operations on edges (because the ART shows that this condition always holds at the particular node). For program LOCKS we see that the loop has been unfolded once, and the test for equality (`i-lastLock == 2`) could be removed.

4 Program Validation

The transformed program is given to the consumer and ready for use. The consumer wants to ensure that the program he/she uses really adheres to the safety property. For this, the consumer can use a computationally inexpensive data flow analysis; inexpensive because – unlike standard DFAs – the information needed can be computed without an iterative fixpoint computation.

Algorithm 1 shows the overall procedure. The objective is here to – again – build a product of program and property automaton, however, this time considering no data states at all. The algorithm just computes for every location of

Algorithm 1. Data-flow analysis as used to validate the conformance of a given program to a given protocol property.

Input: Program $P = ((L, G), l_0)$, protocol automaton $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$
Output: Mapping $m : L \to S \cup \{\bot, \top\}$

```
 1: for all l ∈ L do
 2:    m(l) := ⊥;
 3: m(l_0) := s_0;
 4: stack.push(l_0);
 5: while stack is not empty do
 6:    l := stack.pop();
 7:    for all (l, op, l') ∈ G do
 8:       if op ∈ Σ then
 9:          s' := δ(m(l), op);
10:       else
11:          s' := m(l);
12:       if m(l') = ⊥ then
13:          m(l') := s';
14:          stack.push(l');
15:       else if m(l') ≠ s' then
16:          return {l ↦ ⊤ | l ∈ L};
17: return m;
```

the program the state the property automaton is in when reaching this location. The outcome is a mapping m from locations to states plus the special values $\bot$ (no state assigned to location yet) and $\top$ (more than one state assigned to the location). For arbitrary programs, we could get a *set* of automaton states for a location. However, when the program is the result of our transformation, every location has exactly one associated property automaton state: if the location of the program is derived from ART node (l, s, φ), then $m(l) = s$. Furthermore, given the original program was correct, the state $m(l)$ is never s_{err} (the error state of the property automaton). The algorithm thus determines by a depth-first traversal automaton states for program locations. Once it finds that a second state needs to be added to the states for a location (line 15), it stops.

Lemma 1. *Algorithm 1 has a worst case execution time of* $\mathcal{O}(|L| + |G|)$.

Proof. It is a depth-first search (no fixpoint iteration involved).

The consumer executes this algorithm on the program gained from the producer. If the algorithm terminates with (1) $m(l) \neq \top$ and (2) $m(l) \neq s_{err}$ for all locations l, the program is safe. If (1) does not hold, the program is not the result of our transformation, and either the producer has not followed the transformation scheme or someone else has changed the program after transformation. If (2) does not hold, the producer has given an unsafe program. Both properties are thus detectable. Hence this new approach remains tamper-proof.

We next actually show these properties. The next lemma proves soundness of the algorithm, namely that it computes an overapproximation of the set of

states of the property automaton that the program can be in when in a particular location.

Lemma 2 (Soundness). *The mapping $m : L \to S \cup \{\bot, \top\}$ returned by Algorithm 1 is a overapproximation of the states the property automaton can be in for each program location. That is, for program $P = ((L, G), l_0)$ and protocol automaton $A_{prop} = (\Sigma, S, s_0, s_{err}, \delta)$, if $(l_0, c_0) \overset{op_1}{\to} \ldots \overset{op_n}{\to} (l_n, c_n)$ then $\delta^*(s, (op_1, \ldots, op_n) \restriction \Sigma) = m(l_n)$ or $m(l_n) = \top$.*

Proof. Let $(l_0, c_0) \overset{op_1}{\to} \ldots \overset{op_n}{\to} (l_n, c_n)$ be some path in P. Furthermore, assume $\delta^*(s, (op_1, \ldots, op_n) \restriction \Sigma) \neq m(l_n)$. Furthermore, assume that $m(l) = \top$ for all $l \in L$ does not hold (then $m(l) \neq \top$ for all $l \in L$ by construction of the algorithm). Then, there is a first element in the sequence of locations of the path l_i ($i \in \{1, \ldots, n\}$) such that $\delta^*(s, (op_1, \ldots, op_i) \restriction \Sigma) \neq m(l_i)$ and $\delta^*(s, (op_1, \ldots, op_{i-1}) \restriction \Sigma) = m(l_{i-1})$. Since $m(l_{i-1}) \neq \bot$ and $m(l_{i-1}) \neq \top$, there is an iteration of the while-loop in which l_{i-1} is at the top of the stack. Moreover, as all operations op with $(l_{i-1}, op, \cdot) \in G$ are considered in the inner for-loop of the while-loop, the edge $(l_{i-1}, op_i, l_i) \in G$ is also considered. By construction of the algorithm we then get $s' = \delta(m(l_{i-1}), op_i)$ if $op_i \in \Sigma$ or $s' = m(l_{i-1})$ otherwise. In both cases we thus have $s' = \delta^*(s, (op_1, \ldots, op_i) \restriction \Sigma)$. By construction of the algorithm and by the assumption that $m(l) = \top$ for all $l \in L$ does not hold, we get $m(l_i) = s' = \delta^*(s, (op_1, \ldots, op_i) \restriction \Sigma)$ after the respective iteration of the while-loop. This is in contradiction to $m(l_i) \neq \delta^*(s, (op_1, \ldots, op_i) \restriction \Sigma)$ and not $m(l) = \top$ for all $l \in L$, as the value of $m(l)$ for a given location l is never changed in the algorithm once $m(l) \neq \bot$. □

The second lemma shows preciseness of the algorithm: whenever value $\top$ is computed for some location, then there are at least two syntactically feasible paths of the program which reach different states in the property automaton. In addition, a property automaton state s such that $m(l) = s$ can – at least syntactically – be reached.

Lemma 3 (Preciseness). *Let $m : L \to S \cup \{\bot, \top\}$ be the mapping returned by Algorithm 1. Let $l \in L$. If $m(l) = \top$ then there are two syntactically feasible paths $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$, $l_0 \overset{op'_1}{\to} \ldots \overset{op'_m}{\to} l'_m$ with $l'_m = l_n$ such that $\delta^*(s_0, (op_1, \ldots, op_n) \restriction \Sigma) \neq \delta^*(s_0, (op'_1, \ldots, op'_m) \restriction \Sigma)$. If $m(l) \in S$, then there is a syntactically feasible path $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$ such that $\delta^*(s_0, (op_1, \ldots, op_n)) = m(l)$.*

Proof. Let $m(l) = \top$. Thus, Algorithm 1 terminated in line 16 (rather than 17). Let l'_{m-1}, l'_m denote the locations l and l', respectively, as considered in the last iteration of the while-loop. By construction, we have a path $l_0 \overset{op_1}{\to} \ldots \overset{op_n}{\to} l_n$ such that $l_n = l'_m$ and $\delta^*(s_0, (op_1, \ldots, op_n) \restriction \Sigma) = m(l_n)$ (where m refers to the mapping in the algorithm prior returning $\{l \mapsto \top \mid l \in L\}$). Similar, there is a path $l_0 \overset{op'_1}{\to} \ldots \overset{op'_m}{\to} l'_m$ such that $\delta^*(s_0, (op'_1, \ldots, op'_m) \restriction \Sigma) = s'$. Because the algorithm returns in line 16, we have $m(l_n) \neq s'$ and thus $\delta^*(s_0, (op_1, \ldots, op_n) \restriction \Sigma) \neq \delta^*(s_0, (op'_1, \ldots, op'_m) \restriction \Sigma)$. □

Finally, together these give us our desired result: if the original program satisfies the property specified in the protocol automaton, then so does the transformed program and we can check for this in time linear in the size of the transformed program.

Theorem 2. *Let P be a program. Let $P' = program\_of(T)$, where $T = (N, G, C)$ is an ART for P wrt. A_{prop}. Let $P \models A_{prop}$. We have:*

(a) $P' \models A_{prop}$,
(b) $P' \models A_{prop}$ *is verifiable in* $\mathcal{O}(|N| + |G|)$.

Proof. By Theorem 1 we have $traces(P) = traces(P')$, thus $P' \models A_{prop}$ follows directly from $P \models A_{prop}$. To show that $P' \models A_{prop}$ is verifiable in $\mathcal{O}(|N| + |G|)$ we show that Algorithm 1 can prove $P' \models A_{prop}$ and has an runtime bound of $\mathcal{O}(|N|+|G|)$. The latter directly follows by Lemma 1. For the former we first show that every syntactically feasible path P' ending in l' yields the same automaton state s that is also associated to $l' = (l, s, \varphi)$. Let $l'_0 \overset{op_1}{\rightarrow} \ldots \overset{op_n}{\rightarrow} l'_n$ be some syntactically feasible path in P'. Let (l_i, s_i, φ_i) denote the to l'_i associated ART nodes ($i \in \{0, \ldots, n\}$). By induction hypothesis we have $\delta^*(s_0, (op_1, \ldots, op_{n-1}) \restriction \Sigma) = s_{n-1}$. We have either $(l_{n-1}, op_n, l_n) \in G$ or $(l_{n-1}, op_n, l_{im}) \in G$, $(l_{im}, l_n) \in C$. In both cases, by soundness of the abstractions in the ART, we have $s_n = \delta^*(s_{n-1}, (op_n) \restriction \Sigma)$. Next, by Lemma 3 it follows that Algorithm 1 returns a mapping m with $m(l) \neq \top$ for all $l \in N$ when applied to P' and A_{prop}. By Lemma 3 we also get that $m(l) \neq s_{err}$ for all $l \in L$, as $m(l) = s_{err}$ would require a syntactic path in the ART that leads to the error state which contradicts $P \models A_{prop}$. Finally, by Lemma 2 we get that if $m(l) \in S \setminus \{s_{err}\}$ or $m(l) = \bot$ holds for all locations $l \in E$, then $P' \models A_{prop}$. □

In summary, we have thus obtained a way of transforming a program into an equivalent one, with the same execution time (or less), on which we can however check our property more efficiently.

5 Experimental Results

For the transformed program we have a linear time algorithm for checking adherence to the property. Still, we can of course not be sure that this brings us any improvement over a predicate analysis on the original program since the transformed program is usually larger than the original one. Thus, the purpose of the experiments was mainly to see whether the desired objective of supplying the consumer with a program which is easy to validate is actually met.

We implemented our technique in the program-analysis tool CPAchecker [8]. We chose the Adjustable-Block Encoding [4] predicate analysis. Additionally, we implemented the (consumer-side) validation of the transformed program as a small program written using the compiler construction toolkit Eli [17] that performs parsing and the data-flow analysis described above. All experiments were performed on a 64 bit Ubuntu 12.10 machine with 6 GB RAM, an Intel

i7-2620M at 2.7 GHz CPU, and OpenJDK 7u9 (JVM 23.2-b09). The analysis time does not include the startup time of the JVM and CPAchecker itself (around 1-2 s). The memory usage of the DFA program was measured using the tool Valgrind. The amount of operations in the program can be estimated by looking at the lines of code (LOC) because our transformation to C code inserts a newline at least for every statement.

Beside the basic approach described so far, our implementation contains a further optimization to keep the size of the transformed program small. Instead of performing a single predicate analysis and ART generation only, we perform two passes: the first pass generates the product construction of the CFA and the property automaton yielding ART T_1, and the second pass performs a separate predicate analysis yielding ART T_2. In other words, T_1 is the result of performing the predicate analysis with an empty set of predicates. As a last step, we traverse T_1 and remove every (error) node $(l, s_{err}, true)$ ($true$ represents the empty predicate) such that no (l, s_{err}, φ) appears in T_2 for any φ. Since the program is usually correct (unless the producer tries to sell an incorrect program), this usually means that we remove all nodes in T_1 in which the error state of the protocol automaton occurs. T_1 is then subject to the generation of the new program. The correctness of this step is based on the fact that if the predicate analysis proved that the combination of location and protocol error state is not reachable at all, we can deduce that it must also be unreachable in T_1, because T_1 and T_2 are both overapproximating the program behaviour.

Table 1 shows the benchmark results[1] for different kinds of programs and different analysis'. Except for the lock example from above, the programs in Table 1 were taken from the benchmark set of the Software Verification competition 2012[2]. The token ring benchmark was initially a SystemC program passing tokens between different simulated threads that was amended with a scheduler. The ssl_server (named s3_srvr) benchmark is an excerpt from the OpenSSL project mimicking a server-side SSL connection handler. For the s3_srvr program, we derived different protocol automata of varying complexity. The protocol automaton in case of the token ring benchmark simply checks whether every task is started at most once by the scheduler at the same time.

The *predicate analysis* step in the table includes the predicate analysis that would need to be performed by the code consumer to check correctness of the original, untransformed program. Instead, in our approach this step together with the transformation step is ran by the code producer to generate the C representation of the ART. As the transformation step itself is quite fast (< 300 ms), times were omitted in the table. Afterwards, the code consumer can carry out the cheap *DFA analysis* to show that the transformed program obeys the specified protocol property. To undermine our claim that this technique is an

[1] All benchmark files necessary to reproduce the results are available in the public repository of CPAchecker under `https://svn.sosy-lab.org/software/cpachecker/branches/runtime_verification/`

[2] See `http://sv-comp.sosy-lab.org/`

alternative (in fact, a much better alternative) to proof-carrying code approaches, we also implemented a PCC version based on predicate analysis. The PCC technique attaches the proof in form of the ART and its predicates as a certificate to the program. The *PCC analysis* thus needs to check that the ART is a valid ART and an abstraction of the state space of the program. This in particular necessitates a large number of entailment checks carried out by an SMT solver. However, it avoids abstraction refinement and thus is faster than the original prediate analysis. Still, we see that the speed-up obtained by our transformation based approach is much larger than that of the PCC approach.

Table 1. Benchmark results for different programs and properties

Program (Monitor)	Orig.	Predicate Analysis		PCC Checking		Transf.	DFA Checking	
	LOC	Time	Mem.	Time	Mem.	LOC	Time	Mem.
lock2.c	32	0.066s	88MB	0.068s	88 MB	44	0.00s	0.047 MB
token_ring.02.cil.c	596	7.052s	450MB	1.919s	112 MB	3976	0.01s	0.420 MB
token_ring.03.cil.c	724	14.644s	687MB	5.803s	271 MB	13721	0.01s	1.329 MB
token_ring.04.cil.c	846	83.585s	1139MB	14.281s	592 MB	42944	0.04s	4.121 MB
s3_srvr.cil.c (mon1)	861	25.400s	669MB	5.802s	244 MB	5112	0.00s	0.557 MB
s3_srvr.cil.c (mon2)	861	94.393s	646MB	6.476s	282 MB	4267	0.00s	0.471 MB
s3_srvr.cil.c (mon3)	861	10.616s	484MB	2.701s	158 MB	4267	0.00s	0.471 MB
s3_srvr.cil.c (mon4)	861	20.075s	569MB	5.527s	247 MB	4267	0.01s	0.473 MB
s3_srvr.cil.c (mon5)	861	52.755s	848MB	12.593s	540 MB	7623	0.01s	0.794 MB
s3_srvr.cil.c (mon6)	861	7.385s	390MB	2.926s	138 MB	9279	0.01s	0.999 MB

In summary, we have shown that the consumer-side validation step (DFA) is sped up compared to the analysis step by three to four orders of magnitude[3] (except for the small locks example where the predicate analysis is rather trivial). This significantly demonstrates the usefulness of our approach despite the program size increase of about two to three orders of magnitude. We have also compared the compilation time of the transformed C program with that of the original program to see whether the program size brings compilation times into unwanted ranges: while it increases, it never exceeds one second, thus it stays at an acceptable level.

6 Conclusion

In this paper we have proposed an alternative way of providing the safe execution of untrusted code for code consumers, without imposing the overhead of an extensive verification on the consumer. The approach is based on automatic verification via model checking, more precisely predicate analysis, as the first step for a transformation of the program into a more efficiently analysable program, however, with the same behaviour and performance. For the code consumer

[3] An analysis time of 0.00 seconds means non-measurable amount of time.

correctness can then be proven using a simple data flow analysis. Experimental results have shown that – despite the program usually getting larger during transformation – the analysis can still be significantly simplified.

Related Work. Proof carrying code (PCC) techniques often aim at showing memory and type safety of programs. The PCC approach most closest to ours is that of [18,19] who also use predicate analysis as a basis. In contrast to us they follow a classical PCC approach: the producer generates inductive invariants of the program (which show particular properties) and the validation of the consumer consists of actually checking that the invariant is an invariant. As this involves the call of a theorem prover it will in general take more time than a simple data flow analysis. An exact comparison with our approach was not possible as the programs analysed in [18] are currently not publicly available.

Similar to our approach, some techniques convey the fix-point solution of some iterative algorithm (e.g. abstract interpretation) and check this fix-point using a cheap analysis on the consumer side. Instantiations of this concept can be seen for e.g. Java bytecode [23]. Furthermore, to decrease the amount of memory needed for verification, another approach simplifies the Java bytecode before it is passed to the consumer [20]. In both cases, the class of checkable properties does not amount to safety properties like in our case but is limited to simple type properties on the bytecode level.

The idea of using the abstract reachability tree of a program obtained by a predicate analysis is also the idea underlying conditional model checking [6]. While the transformation of an ART into a program is generally envisaged (for the purpose of benchmark generation), the approach focuses on generating conditions for use in further verification runs. Generation of programs from parts of an ART, namely certain counterexamples, is also the basis of the work on *path invariants* [7] which presents an effective way of abstraction refinement.

In the area of model checking, *slicing* is an established technique for reducing the size of program before verification [14]. This transformation is however not generally behaviour preserving, but only property preserving.

A lot of works focus on verifying protocol-like properties of programs. This starts as early as the eighties with the work on typestate analysis [24]. Typestate is a concept which enhances types with information about their state and the operations executable in particular states. Recent approaches have enhanced typestate analysis with ideas of predicate abstraction and abstraction refinement [12,11]. Others use an inconclusive typestate analysis to generate residual monitors to be used in runtime verification [15,9]. This can be seen as a form of partial evaluation. The tool SLAM [2] uses predicate abstraction techniques to analysis C programs with respect to typestate like properties.

Future Work. In the future, we intend to investigate other instantiations of our general framework, first of all for other property classes (e.g. memory safety). We believe that the basic principle of having a model checker do a path-sensitive analysis constructing an ART, which is transformed into a program on which a single-pass data-flow analysis can then prove the desired property, is applicable to a large number of properties since the construction of the ART unfolds the

program in such a way that every node in the ART is "unique" wrt. program location and property of interest. We furthermore aim at using this approach for generation of residual monitors, to enhance runtime verification when the property could not be proven on the program. It would furthermore be interesting to investigate whether the source code property of being "easily verifiable" would in some way carry over to the compiled binary.

References

1. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 268–283. Springer, Heidelberg (2001)
2. Ball, T., Rajamani, S.K.: Automatically validating temporal safety properties of interfaces. In: Dwyer, M.B. (ed.) SPIN 2001. LNCS, vol. 2057, pp. 103–122. Springer, Heidelberg (2001)
3. Bauer, L., Schneider, M.A., Felten, E.W., Appel, A.W.: Access control on the web using proof-carrying authorization. In: Proceedings of the 3rd DARPA Information Survivability Conference and Exposition, DISCEX (2), pp. 117–119 (2003)
4. Beyer, D., Keremoglu, M., Wendler, P.: Predicate Abstraction with Adjustable-Block Encoding. In: FMCAD 2010, pp. 189–197 (2010)
5. Beyer, D., Chlipala, A.J., Henzinger, T.A., Jhala, R., Majumdar, R.: The Blast query language for software verification. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 2–18. Springer, Heidelberg (2004)
6. Beyer, D., Henzinger, T.A., Keremoglu, M.E., Wendler, P.: Conditional model checking: a technique to pass information between verifiers. In: Tracz, W., Robillard, M.P., Bultan, T. (eds.) SIGSOFT FSE, p. 57. ACM (2012)
7. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Path invariants. In: Ferrante, J., McKinley, K.S. (eds.) PLDI, pp. 300–309. ACM (2007)
8. Beyer, D., Keremoglu, M.E.: cPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
9. Bodden, E., Lam, P., Hendren, L.: Clara: A Framework for Partially Evaluating Finite-State Runtime Monitors Ahead of Time. In: Barringer, H., Falcone, Y., Finkbeiner, B., Havelund, K., Lee, I., Pace, G., Roşu, G., Sokolsky, O., Tillmann, N. (eds.) RV 2010. LNCS, vol. 6418, pp. 183–197. Springer, Heidelberg (2010)
10. Crary, K., Weirich, S.: Resource bound certification. In: Proceedings of the ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL), pp. 184–198 (2000)
11. Das, M., Lerner, S., Seigle, M.: ESP: Path-Sensitive Program Verification in Polynomial Time. In: PLDI, pp. 57–68 (2002)
12. Dhurjati, D., Das, M., Yang, Y.: Path-sensitive dataflow analysis with iterative refinement. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 425–442. Springer, Heidelberg (2006)
13. Drzevitzky, S., Kastens, U., Platzner, M.: Proof-carrying hardware: Towards runtime verification of reconfigurable modules. In: Proceedings of the International Conference on Reconfigurable Computing (ReConFig), pp. 189–194. IEEE (2009)
14. Dwyer, M.B., Hatcliff, J., Hoosier, M., Ranganath, V.P., Robby, Wallentine, T.: Evaluating the effectiveness of slicing for model reduction of concurrent object-oriented programs. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 73–89. Springer, Heidelberg (2006)

15. Dwyer, M.B., Purandare, R.: Residual dynamic typestate analysis exploiting static analysis: results to reformulate and reduce the cost of dynamic analysis. In: ASE, pp. 124–133. ACM (2007)
16. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997)
17. Gray, R., Levi, S., Heuring, V., Sloane, A., Waite, W.: Eli: A complete, flexible compiler construction system. Communications of the ACM 35(2), 121–130 (1992)
18. Henzinger, T.A., Jhala, R., Majumdar, R., Necula, G.C., Sutre, G., Weimer, W.: Temporal-safety proofs for systems code. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 526–538. Springer, Heidelberg (2002)
19. Henzinger, T.A., Jhala, R., Majumdar, R., Sanvido, M.A.A.: Extreme model checking. In: Dershowitz, N. (ed.) Verification: Theory and Practice. LNCS, vol. 2772, pp. 332–358. Springer, Heidelberg (2004)
20. Leroy, X.: Bytecode verification on java smart cards. Softw. Pract. Exper. 32(4), 319–340 (2002)
21. Necula, G.C.: Proof-carrying code. In: POPL 1997, pp. 106–119. ACM, New York (1997)
22. Necula, G.C., Lee, P.: Safe, untrusted agents using proof-carrying code. In: Vigna, G. (ed.) Mobile Agents and Security. LNCS, vol. 1419, pp. 61–91. Springer, Heidelberg (1998)
23. Rose, E.: Lightweight bytecode verification. Journal of Automated Reasoning 31, 303–334 (2003)
24. Strom, R.E., Yemini, S.: Typestate: A programming language concept for enhancing software reliability. IEEE Trans. Software Eng. 12(1), 157–171 (1986)

PARTY
Parameterized Synthesis of Token Rings*

Ayrat Khalimov, Swen Jacobs, and Roderick Bloem

Graz University of Technology, Austria

Abstract. Synthesis is the process of automatically constructing an implementation from a specification. In parameterized synthesis, we construct a single process such that the distributed system consisting of an arbitratry number of copies of the process satisfies a parameterized specification. In this paper, we present Party, a tool for parameterized synthesis from specifications in indexed linear temporal logic. Our approach extends SMT-based bounded synthesis, a flexible method for distributed synthesis, to parameterized specifications. In the current version, Party can be used to solve the parameterized synthesis problem for token-ring architectures. The tool can also synthesize monolithic systems, for which we provide a comparison to other state-of-the-art synthesis tools.

1 Introduction

Synthesis methods and tools have received increased attention in recent years, as suitable solutions have been found for several synthesis tasks that have long been considered intractable. Although current tools have made large strides in efficiency, the run time of synthesis tools and the size of the resulting system still depend strongly on the size of the specification. In the case of parameterized specifications, this is particularly noticeable, and often unnecessary.

Bounded synthesis [13] is a method for solving the LTL synthesis problem by considering finite-state implementations with bounded resources. The space of all possible implementations is explored by iteratively increasing the bound. While several tools for LTL synthesis are based on variants of this approach [9,3], none of the available tools support distributed or parameterized synthesis.

In this paper, we introduce Party, the first tool that implements parameterized synthesis [14], based on the original SMT-based approach to bounded synthesis. Using cutoff results from parameterized verification [10], parameterized synthesis problems are reduced to distributed synthesis problems, and solved by bounded synthesis. While mainly intended to solve parameterized synthesis problems, Party can also be used for standard (monolithic) synthesis tasks.

Party is available at `https://github.com/5nizza/Party`. In the following sections, we present the background and implementation details of Party, and experimental results comparing Party to existing synthesis tools.

* This work was supported in part by the European Commission through project DIAMOND (FP7-2009-IST-4-248613), and by the Austrian Science Fund (FWF) under the RiSE National Research Network (S11406).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 928–933, 2013.

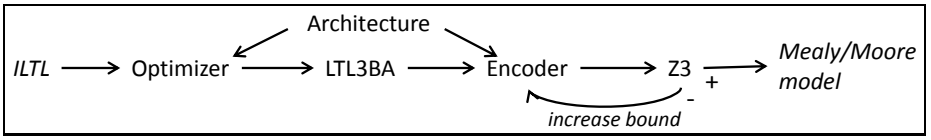


Fig. 1. The PARTY high-level control flow

2 Background: Parameterized Synthesis

We use an approach that reduces the problem of LTL synthesis to a sequence of first-order satisfiability problems. This reduction, called *Bounded Synthesis*, has been introduced by Schewe and Finkbeiner [13] for the case of distributed systems with a fixed number of finite-state components. It is based on a translation of the specification into a universal co-Büchi automaton, followed by the generation of a first-order constraint that quantifies over an unbounded number of states. The existence of a solution to this constraint is equivalent to the realizability of a system accepted by the automaton, and thus synthesis is reduced to a satisfiability problem. To make this problem decidable, an additional bound on the size of the implementation is asserted, resulting in a semi-decision procedure for the problem by considering increasing bounds.

Based on results by Emerson and Namjoshi [10] on model checking parameterized token rings, Jacobs and Bloem [14] extended the bounded synthesis method to parameterized systems, consisting of an arbitrary number of isomorphic processes. Depending on the syntactic form of the specification, synthesis of parameterized token rings can be reduced to synthesis of isomorphic processes in small rings, and the resulting implementations are guaranteed to satisfy the specification in rings of arbitrary size. The sufficient size of the ring is called the *cutoff* and should not be confused with the *bound* on size of individual processes.

In previous work [16], we used results by Clarke et al. [5] to extend the applicability of this approach, and generalized optimizations from parameterized or distributed verification to synthesis methods, to improve its efficiency in practice.

While the approaches above have been implemented in a prototype before [16], in PARTY we have added several features such as the synthesis of Mealy machines, as well as the synthesis of non-parameterized, monolithic, systems. Also, we have refactored our implementation for usability, including an input language that is derived from the language used by Acacia+.

3 Tool Description

The high-level control flow of PARTY is given in Fig. 1.

Input. Specifications consist of four parts: inputs, outputs, assumptions and guarantees. Assumptions and guarantees are in ILTL and may contain universal quantifiers at the beginning of the expression. This simple structure of properties, $\forall i.\ A_i \rightarrow \forall j.\ G_j$, is enough to model all examples we have considered

thus far. The format of the language is very similar to that of Acacia+; an example specification is given in Listing 1. Note that, although specifications of parameterized token rings are in ILTL\X, Party supports a X operator whose semantics is local to a given process [16].

```
[INPUT_VARIABLES]
r;
[OUTPUT_VARIABLES]
g;
[ASSUMPTIONS]
Forall (i) r_i=0;
Forall (i) G(((r_i=1)*(g_i=0)->X(r_i=1)) *
             ((r_i=0)*(g_i=1)->X(r_i=0)));
Forall (i) G(F((r_i=0)+(g_i=0)));
[GUARANTEES]
Forall (i) g_i=0;
Forall (i,j) G(!((g_i=1) * (g_j=1)));
Forall (i) G((((r_i=0)*(g_i=0))->X(g_i=0)) *
             (((r_i=1)*(g_i=1))->X(g_i=1)));
Forall (i) G(F(((r_i=1)*(g_i=1)) +
               ((r_i=0)*(g_i=0))));
```

Listing 1. Specification of parameterized Pnueli arbiter

Optimizer. *Optimizer* takes as input an ILTL specification and an architecture (currently *token-ring* or *monolithic*). It adds domain-specific environment assumptions, such as fair scheduling, to the specification and optimizes it according to user provided option `-opt (no, strength, async_hub, sync_hub)`. Then, Optimizer identifies the cutoff of the specification and instantiates it accordingly. The instantiated specification is in LTL without quantifiers, and can be synthesized with an adapted version of the bounded synthesis approach.

Encoder, Z3 SMT Solver. The LTL specification obtained from Optimizer is translated into an automaton by LTL3BA [1]. The result is passed to *Encoder*, together with the architecture. Currently Encoder supports monolithic architectures and parameterized token rings, but it can be extended to other parameterized architectures and general distributed synthesis.

Given a bound on the implementation, Encoder generates an SMT query in AUFLIA logic. This query is fed to solver Z3 [6] for a satisfiability check. If the SMT query for a given bound is unsatisfiable, control is returned to the Encoder who increases the bound, encodes a new query and feeds it to the solver. If the query is satisfiable, the model is converted into a Mealy or a Moore machine.

Output. For realizable inputs, Party outputs a Mealy or Moore machine in dot or nusmv format. NuSMV v2.5.4 [4] can be used for model checking.

Implementation. Party is written in python3 (5k lines of code) and tested on Ubuntu 12.04. Python does not introduce a significant overhead since the most computationally expensive parts are done in LTL3BA (v.1.0.1) and Z3 (v.4.3.1).

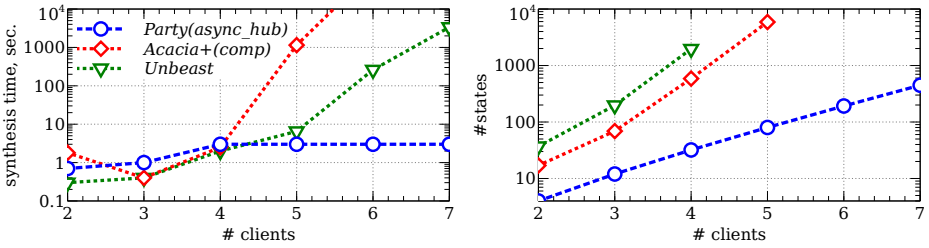


Fig. 2. Synthesis time and model size on a parameterized full arbiter example[16]. The tools demonstrate similar behavior on 'x_arbiter' example and on 'Pnueli' arbiter (but in the latter case async_hub optimization is not complete).

4 Experiments

We compare PARTY with Acacia+(v2.1) and UNBEAST (v0.6b) on parameterized and several monolithic examples.

Parameterized Benchmarks. To test parameterized part of PARTY, we use several arbiter specifications: *full* arbiter [16], *'Pnueli'* arbiter in GR(1) style, *x_arbiter* that uses the local next operator, and variants of these.

Fig. 2 shows the efficiency of parameterized synthesis compared to the standard approach, for 'full' arbiter with a parametric number of clients. Starting with 6 clients, PARTY outperforms the other tools by orders of magnitude. The other tools can only generate arbiters with 5 or 6 clients within one hour, whereas the parameterized approach can stop after synthesizing a token ring of cutoff size 4, and clone the resulting process model to form a ring of any larger size.

The right-hand side of the figure shows implementation size with increasing number of clients. Models generated by PARTY are several orders of magnitude smaller than others and grows less steeply with increasing the number of clients.

Monolithic Benchmarks. For the comparison we use realizable benchmarks from tool Lily (3,5-10,12-23) [15], load balancer(*lb2-lb4*) [9], and genbuf benchmark(*gb2*)[2]. Table 1 compares times of PARTY with times of Acacia+/UNBEAST. All models synthesized by PARTY were model checked with NuSMV [4].

It is difficult to provide a fair comparison due to different semantics of system models. For example, Acacia+ outputs Moore machines, UNBEAST Mealy machines as NuSMV models, and PARTY supports both. Also, UNBEAST has its own xml-based input format and does not provide a converter from other formats. Therefore, we could not run UNBEAST on *gb2* benchmark.

The difficulty of fair comparison is reflected by Table 1. The tools were run with their default parameters. UNBEAST was run with extracting models option. Acacia+ and PARTY were provided with two different specifications: in Moore semantics (the system moves first), and in Mealy semantics (the environment moves first). In the second case PARTY generated Mealy models, while Acacia+ still generated Moore-like models. The table shows that UNBEAST outperforms other tools in terms of synthesis time, and models in PARTY are the smallest.

Table 1. Comparison of PARTY monolithic with Acacia+ and UNBEAST (t/o=1h). Numbers in parenthesis mean the size of implementation[1].

	lily	lily16	lb2	lb3	lb4	gb2
UNBEAST(Mealy)	4(540/160)	0.1(55/54)	0.2(25/11)	1(576/33)	13(m/o)	-
Acacia+(Mealy)	12(80)	1(15)	2(10)	1(42)	54(145)	-
PARTY(Mealy)	22(33)	35(6)	1(1)	3(2)	139(3)	-
Acacia+(Moore)	7(61)	1(15)	1(7)	4(14)	1639(41)	1(49)
PARTY(Moore)	12(40)	1526(8)	1(3)	364(6)	t/o	t/o

A detailed analysis shows that PARTY spends most of the time in SMT solving, where in turn proving unsatisfiability for bounds that are too small takes most of the time. For example, the synthesis time for *lily16* is 25 minutes if we explore all model sizes starting with 1. But if we force PARTY to search the model of exact size 8, the solution is found in 2 minutes. This means that we need to explore incremental solving and other methods to avoid long unsatisfiability checks.

Incremental solving. PARTY supports two incremental approaches: for increasing size of rings (parameterized architectures only), and for increasing bounds.

Instead of searching for a model in a token ring of the cutoff size, we can use even smaller rings and then check if the result satisfies the specification in a ring of sufficient size. Preliminary experiments demonstrate the effectiveness of this approach: the 'Pnueli' arbiter can be synthesized in 50 seconds using the usual approach vs. 15 seconds using the incremental approach.

To handle increasing bounds, we can use incrementality of SMT solvers when unsatisfiability for the current bound is detected. When the bound is increased, we pull constraints directly relating to the old bound, and push new ones. On most benchmarks, this approach is comparable to non-incremental one, but on some examples it is much faster: full3: 506 seconds (orig) vs. 140 seconds (incr).

5 Conclusions

We presented PARTY, a tool for parameterized synthesis. In the current version, the tool can synthesize Mealy and Moore machines as process implementations in monolithic architectures or parameterized token-ring architectures. For the latter case, it implements optimizations that speed up synthesis by several orders of magnitude. The input language is derived from languages of existing tools, supports full LTL for monolithic and ILTL\X for parameterized architectures.

Besides the fact that this is the first implementation of parameterized synthesis, an experimental comparison to other synthesis tools has been difficult

[1] UNBEAST model sizes are calculated by NuSMV with `-r` option as suggested by Rüdiger Ehlers. Two sizes are given: for default model extraction method, and for a learning based method [8]. On *lb4* NuSMV crashed with a memory allocation error.

because tools are often specialized to a subclass of problems, and use different input languages (see also Ehlers [7]). It may be worthwhile to discuss standards for languages and subclasses of synthesis problems, much like in other automated reasoning communities, e.g., represented by the SMT-LIB initiative.

PARTY is designed modularly, and we are working on several extensions. Most importantly, parameterized synthesis can be extended to other architectures that allow automatic detection of cutoffs [5,11]. Also, we plan to further increase efficiency of the synthesis process, either by more high-level optimizations or by integration of verification techniques, like the lazy synthesis approach [12].

References

1. Babiak, T., Křetínský, M., Řehák, V., Strejček, J.: LTL to büchi automata translation: Fast and more deterministic. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 95–109. Springer, Heidelberg (2012)
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. ENTCS 190(4), 3–16 (2007)
3. Bohy, A., Bruyère, V., Filiot, E., Jin, N., Raskin, J.-F.: Acacia+, a tool for LTL synthesis. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 652–657. Springer, Heidelberg (2012)
4. Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking (July 2002)
5. Clarke, E., Talupur, M., Touili, T., Veith, H.: Verification by network decomposition. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 276–291. Springer, Heidelberg (2004)
6. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
7. Ehlers, R.: Experimental aspects of synthesis. In: Proceedings of the International Workshop on Interactions, Games and Protocols. EPTCS, vol. 50 (2011)
8. Ehlers, R., Könighofer, R., Hofferek, G.: Symbolically synthesizing small circuits. In: Cabodi, G., Singh, S. (eds.) FMCAD, pp. 91–100. IEEE (2012)
9. Ehlers, R.: Unbeast: Symbolic bounded synthesis. In: Abdulla, P.A., Leino, K.R.M. (eds.) TACAS 2011. LNCS, vol. 6605, pp. 272–275. Springer, Heidelberg (2011)
10. Emerson, E.A., Namjoshi, K.S.: On reasoning about rings. Int. J. Found. Comput. Sci. 14(4), 527–550 (2003)
11. Emerson, E.A., Kahlon, V.: Reducing model checking of the many to the few. In: McAllester, D. (ed.) CADE 2000. LNCS, vol. 1831, pp. 236–254. Springer, Heidelberg (2000)
12. Finkbeiner, B., Jacobs, S.: Lazy synthesis. In: Kuncak, V., Rybalchenko, A. (eds.) VMCAI 2012. LNCS, vol. 7148, pp. 219–234. Springer, Heidelberg (2012)
13. Finkbeiner, B., Schewe, S.: Bounded synthesis. International Journal on Software Tools for Technology Transfer, 1–21 (2012)
14. Jacobs, S., Bloem, R.: Parameterized synthesis. In: Flanagan, C., König, B. (eds.) TACAS 2012. LNCS, vol. 7214, pp. 362–376. Springer, Heidelberg (2012)
15. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society (2006)
16. Khalimov, A., Jacobs, S., Bloem, R.: Towards efficient parameterized synthesis. In: Giacobazzi, R., Berdine, J., Mastroeni, I. (eds.) VMCAI 2013. LNCS, vol. 7737, pp. 108–127. Springer, Heidelberg (2013)

Recursive Program Synthesis

Aws Albarghouthi[1], Sumit Gulwani[2], and Zachary Kincaid[1]

[1] University of Toronto
[2] Microsoft Research

Abstract. Input-output examples are a simple and accessible way of describing program behaviour. Program synthesis from input-output examples has the potential of extending the range of computational tasks achievable by end-users who have no programming knowledge, but can articulate their desired computations by describing input-output behaviour. In this paper, we present Escher, a generic and efficient algorithm that interacts with the user via input-output examples, and synthesizes *recursive programs* implementing intended behaviour. Escher is parameterized by the *components* (instructions) that can be used in the program, thus providing a generic synthesis algorithm that can be instantiated to suit different domains. To search through the space of programs, Escher adopts a novel search strategy that utilizes special data structures for inferring conditionals and synthesizing recursive procedures. Our experimental evaluation of Escher demonstrates its ability to efficiently synthesize a wide range of programs, manipulating integers, lists, and trees. Moreover, we show that Escher outperforms a state-of-the-art SAT-based synthesis tool from the literature.

1 Introduction

Program synthesis from specifications is a foundational problem that crosses the boundaries of formal methods, software engineering, and artificial intelligence. Traditionally, specifications written in logics (such as first-order and temporal logics) have been used to synthesize programs, e.g., [16,17]. More recently, we have witnessed renewed interest in the program synthesis question, and a shift from the traditional logical specifications to specifications presented as input-output examples, e.g., [8,13,11,15,12]. One of the main advantages of synthesis from input-output examples is that it extends the user base of synthesis techniques from algorithm and protocol designers to end-users who have no programming knowledge, but can articulate their desired computational tasks as input-output examples. For instance, recent work on synthesizing string manipulation programs from examples in spreadsheets [8] has already made the transition from research into practice, as seen in the latest release of Microsoft Excel [1]. These synthesis techniques capitalize on the fact that end-users are often interested in performing simple operations within specific domains (e.g., string manipulation), and can easily supply the synthesizer with examples demonstrating the tasks they wish to perform.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 934–950, 2013.

In this paper, our goal is to provide a *generic* and *efficient* synthesis algorithm that interacts with users via input-output examples, and allows for synthesis of a wide range of programs. To that end, we present ESCHER, an inductive synthesis algorithm that learns a *recursive procedure* from input-output examples provided by the user. ESCHER is parameterized by the set of *components* (instructions) that can appear in the synthesized program. Components can include basic instructions like integer addition or list concatenation, API calls, and recursive calls (i.e., instructions calling the synthesized program). This allows ESCHER to be instantiated with different sets of components and applied to different domains (e.g., integer or list manipulation). ESCHER assumes the existence of an *oracle*, simulating the user, which given an input returns an output. By interacting with the oracle, ESCHER is able to synthesize recursive programs comprised of a given set of components.

To search through the space of programs, ESCHER adopts an explicit search strategy that *alternates* between two phases: (1) *Forward Search*: in the forward search phase, ESCHER enumerates programs by picking a synthesized program and augmenting it with new components. For example, a program $f(x)$, which applies a component f to the input x, can be extended to $g(f(x))$. (2) *Conditional Inference*: in the conditional inference phase, ESCHER utilizes a novel data-structure, called a *goal graph*, which enables detecting when two programs synthesized by the forward search have to be joined by a conditional statement. For example, two programs $f(x)$ and $g(x)$ can be used to construct **if** $c(x)$ **then** $f(x)$ **else** $g(x)$.

The power of our search strategy is two-fold: (1) By alternating between forward search and conditional inference, we generate conditionals on demand, i.e., when the goal graph determines they are needed. This is in contrast to other component-based techniques, e.g., [10,13], that use an explicit if-then-else component. (2) By adopting an explicit search strategy, as opposed to an SMT encoding like [23,10], we do not restrict the range of synthesizable programs by the supported theories. Moreover, our explicit search strategy allows us to easily apply search heuristics, e.g., biasing the search towards synthesizing smaller programs. We have used ESCHER to synthesize a wide range of programs, manipulating integers, lists, as well as trees. Our experimental results indicate the efficiency of ESCHER and the power of our design choices, including the *goal graph* data structure for conditional inference. Furthermore, we compare ESCHER with SKETCH [23], a state-of-the-art synthesis tool, and show how ESCHER's search technique outperforms SKETCH's SAT-based technique for synthesizing programs from components.

Contributions. We summarize our contributions as follows: (1) ESCHER: a novel algorithm for synthesizing recursive programs that (a) interacts with users via input-output examples to learn programs; (b) is parameterized by the set of components allowed to appear in the program, thus providing a generic synthesis technique; and (c) uses new techniques and data structures for searching through the space of programs. (2) An implementation and an evaluation of ESCHER on a set of benchmarks that demonstrate its effectiveness at synthesizing a wide range

of recursive programs. Moreover, our results highlight the power of our goal graph data structure for conditional inference. (3) A comparison of Escher with a state-of-the-art synthesis tool from the literature which demonstrates Escher's superiority in terms of efficiency and scalability.

2 Overview

In this section, we illustrate the operation of Escher on a simple example. Suppose the user would like to synthesize a program that counts the number of elements in a list of integers (procedure `length` with input parameter `i`). Suppose also that Escher is instantiated with the following set of components: `inc`, takes an integer and returns its successor; `isEmpty`, takes a list and returns `T` (true) if the list is empty and `F` (false) otherwise; `tail`, takes a list and returns its tail; `zero`, a nullary component representing the constant `0`; and `length`, a component representing the function that we would like to synthesize. The existence of `length` as a component allows the synthesized function to be recursive, by using `length` to simulate the recursive call.

Escher alternates between a *forward search* phase and a *conditional inference* phase. We assume that Escher's alternation is guided by a heuristic function h that maps a program to a natural number, where the lower the number the more desirable the program is. For the sake of illustration, we assume that the value of h is always the size of the program, except when a program uses the same component more than once, in which case it is penalized.

Initially, the user supplies Escher with input-output values on which to conduct the search. Suppose the *input values* are the lists `[]`, `[2]`, and `[1,2]`. We represent input values as a *value vector* ⟨`[],[2],[1,2]`⟩. The desired *goal* value vector (outputs) corresponding to the input values is ⟨`0,1,2`⟩, where `0`, `1`, and `2` are the lengths of the lists `[]`, `[2]`, and `[1,2]`, respectively.

First Alternation. First, in the forward search phase, Escher creates programs that are composed of inputs or nullary components (i.e., programs of size 1). In our case, as shown in Figure 1, these are programs $\texttt{P}_1$ (the program that returns the input `i` – the identity function), and $\texttt{P}_2$ (the program that always returns `0` for all inputs). Note that each program is associated with a value vector representing its valuation, for example, the value vector of $\texttt{P}_1$ is ⟨`[],[2],[1,2]`⟩. Obviously, neither $\texttt{P}_1$ nor $\texttt{P}_2$ satisfy our goal ⟨`0,1,2`⟩. But notice that the value vector of $\texttt{P}_2$, ⟨`0,0,0`⟩, overlaps with our goal ⟨`0,1,2`⟩ in the first position, i.e., produces the correct output for the input `[]`. Therefore, the conditional inference phase determines that one way to reach our goal is to synthesize a program $\texttt{P}_r$ of the form `if` $\texttt{P}_{cond}$ `then` $\texttt{P}_2$ `else` $\texttt{P}_{else}$, where $\texttt{P}_{cond}$ is a program that evaluates to `T` on the input `[]`, and `F` on the inputs `[2]` and `[1,2]`; and $\texttt{P}_{else}$ is a program that evaluates to ⟨`1,2`⟩ on the inputs ⟨`[2],[1,2]`⟩. Intuitively, conditional inference determines that we can return `0` (program $\texttt{P}_2$) for the input `[]`, and synthesize another program to deal with inputs `[2]` and `[1,2]`.

Escher represents this strategy for synthesizing `length` by creating a *goal graph*, as shown in Figure 2(a). The goal graph is a novel data structure

h	Program $\mathtt{P}_i$
1	$\mathtt{P}_1 = \mathtt{i} \rightarrow \langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$
	$\mathtt{P}_2 = \mathtt{zero} \rightarrow \langle \mathtt{0,0,0} \rangle$
2	$\mathtt{P}_3 = \mathtt{tail}(\mathtt{P}_1) \rightarrow \langle \mathtt{err}, \mathtt{[]}, \mathtt{[2]} \rangle$
	$\mathtt{P}_4 = \mathtt{inc}(\mathtt{P}_2) \rightarrow \langle \mathtt{1,1,1} \rangle$
	$\mathtt{P}_5 = \mathtt{isEmpty}(\mathtt{P}_1) \rightarrow \langle \mathtt{T,F,F} \rangle$
3	$\mathtt{P}_6 = \mathtt{length}(\mathtt{P}_3) \rightarrow \langle \mathtt{err,0,1} \rangle$
	$\mathtt{P}_7 = \mathtt{tail}(\mathtt{P}_3) \rightarrow \langle \mathtt{err,err,[]} \rangle$
4	$\mathtt{P}_8 = \mathtt{inc}(\mathtt{P}_7) \rightarrow \langle \mathtt{err,1,2} \rangle$

Fig. 1. Synthesized programs $\mathtt{P}_i$ organized by heuristic value h

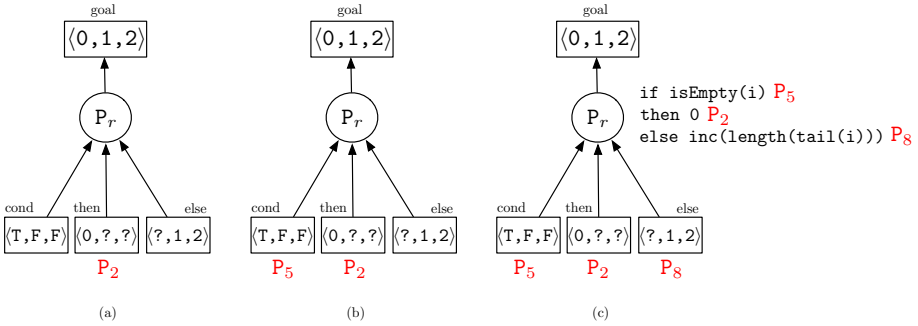


Fig. 2. Goal graph after (a) first, (b) second and third, and (c) final alternations

employed by ESCHER that specifies how to reach the goal $\langle \mathtt{0,1,2} \rangle$ by synthesizing a program $\mathtt{P}_r$. Specifically, we need a program $\mathtt{P}_{cond}$ that returns $\langle \mathtt{T,F,F} \rangle$ for inputs $\langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$, and a program $\mathtt{P}_{else}$ that returns $\langle \mathtt{?,1,2} \rangle$ for inputs $\langle \mathtt{[]}, \mathtt{[2]}, \mathtt{[1,2]} \rangle$, where '?' denotes a "don't care" value, since the first input `[]` will not enter the `else` branch. Note that the `then` branch is already solvable using $\mathtt{P}_2$, and is therefore annotated with it.

Second Alternation. The forward search now synthesizes programs by applying components to programs found for the heuristic value 1. Note that ESCHER can apply `length` (the recursive call) on the input values (program $\mathtt{P}_1$), but this obviously results in a non-terminating program (namely, the program `let length(i) = length(i)`). ESCHER employs a termination argument that detects and discards non-terminating programs. Next, by applying `tail` to $\mathtt{P}_1$, we get the program $\mathtt{P}_3$ that computes $\langle \mathtt{err}, \mathtt{[]}, \mathtt{[2]} \rangle$, where `err` is a special symbol representing an error value, since `tail` is undefined on the empty list. Program $\mathtt{P}_4$ results from applying `inc` to $\mathtt{P}_2$ and generates the value vector $\langle \mathtt{1,1,1} \rangle$. Program $\mathtt{P}_5$ is generated by applying `isEmpty` to $\mathtt{P}_1$, resulting in the value vector $\langle \mathtt{T,F,F} \rangle$. Now, the conditional inference phase discovers that $\mathtt{P}_5$ solves the $\langle \mathtt{T,F,F} \rangle$ subgoal in the goal graph, and annotates it with $\mathtt{P}_5$ (see Figure 2(b)).

In the third alternation, new programs are generated, but none of our goals are solved.

Final Alternation. Finally, forward search applies `inc` to $\mathtt{P}_6$ and generates $\mathtt{P}_8$ with the value vector $\langle \mathsf{err}, 1, 2\rangle$. Conditional inference recognizes that this program solves the subgoal $\langle ?, 1, 2\rangle$. Since all subgoals of our main goal $\langle 0, 1, 2\rangle$ are now solved, we can synthesize the final program `if` $\mathtt{P}_5$ `then` $\mathtt{P}_2$ `else` $\mathtt{P}_8$. Figure 2(c) shows the result produced by ESCHER, which satisfies the given input-output values, and extrapolates to the behaviour intended by the user.

It is important to note that each program is associated with a value vector representing its execution on the given inputs. This allows us to restrict the search space by treating programs with equivalent value vectors as equivalent programs. This *observational equivalence* property can reduce the search space drastically, as will be demonstrated experimentally in Section 4. Moreover, our use of goal graphs allows us to efficiently synthesize programs that contain conditionals. Other component-based techniques like [10,13] approach this problem by using an if-then-else component; in Section 4, we demonstrate the advantages of the goal graph over this approach experimentally.

3 The Escher Algorithm

In this section, we provide the basic definitions required for the rest of the paper, present the ESCHER algorithm and discuss its properties and practical considerations.

3.1 Definitions

Synthesis Task. We define a *synthesis task* $\mathcal{S}$ as a pair $(Ex, Comps)$, where Ex is a list of input-output examples, and $Comps$ is the set of components allowed to appear in the synthesized program. We assume that $Comps$ contains a special component self which is treated as a recursive call to the synthesized program. Every component c is associated with an arity $\mathsf{a}(c) \in [0, \infty)$, indicating the number of input parameters c accepts. We define $\mathsf{a}(\mathsf{self})$ to be the number of input parameters in the examples in Ex. We assume, without loss of generality, that each component returns one value.

We use $\mathcal{V}$ to denote the set of *values* (which may include integers, Booleans, lists of integers, etc.). We make three assumptions about $\mathcal{V}$: equality must be decidable on $\mathcal{V}$, there is a well-founded relation $\prec \subseteq \mathcal{V} \times \mathcal{V}$, and $\mathcal{V}$ must contain a distinguished err value that denotes the result of an erroneous computation (e.g., a run-time exception, a type error, or a non-terminating computation), as well as the Boolean values `T` and `F`. We assume the existence of a total function eval that takes a component c and an $\mathsf{a}(c)$-tuple of values $I \in \mathcal{V}^{\mathsf{a}(c)}$, and returns a value representing the result of applying c to I. For example, if c is integer division, then $\mathsf{eval}(c, (6, 3)) = 2$ and $\mathsf{eval}(c, (4, 0)) = \mathsf{err}$. For recursive program synthesis, we treat the evaluation of the target component self as a call to the *Oracle* (that is, self is evaluated by the user). We memoize such calls to avoid making repeated queries to the *Oracle* for the same input.

Programs. Given a synthesis task $(Ex, Comps)$, we define a *program* P over $Comps$ using the following grammar:

$$
\begin{array}{rcll}
P \in \mathit{Program} & ::= & \texttt{if } P_{cond} \texttt{ then } P_{then} \texttt{ else } P_{else} & \\
& | & c(P_1, \ldots, P_{\mathsf{a}(c)}) & \text{if } \mathsf{a}(c) > 0 \\
& | & c & \text{if } \mathsf{a}(c) = 0 \\
& | & x_j & 1 \leq j \leq \mathsf{a}(\mathsf{self})
\end{array}
$$

where x_j is an input parameter, and c is a component. We use *Vars* to denote the set of all input parameters.

Note that our programming language is untyped (programs encountering a run-time type error evaluate to err), first order (although the components themselves may be higher-order), and purely functional.

3.2 Algorithm Description

Problem Definition. We start by formalizing the synthesis problem as follows. We use (in_i, out_i) to denote the ith input-output example in Ex (where $in_i, out_i \in \mathcal{V}$), $v[j]$ to denote the jth element of a value vector v, and define a function $\mathsf{eval}_v : \mathit{Program} \to \mathcal{V}^{|Ex|}$ as follows:

$$\mathsf{eval}_v(x_j)[i] = in_i[j]$$
$$\mathsf{eval}_v(c)[i] = \mathsf{eval}(c, ())$$
$$\mathsf{eval}_v(c(P_1, \ldots, P_n))[i] = \mathsf{eval}(c, (\mathsf{eval}_v(P_1)[i], \ldots, \mathsf{eval}_v(P_n)[i]))$$
$$\mathsf{eval}_v(\texttt{if } P_{cond} \texttt{ then } P_{then} \texttt{ else } P_{else})[i] = \begin{cases} \mathsf{eval}_v(P_{then})[i] & \text{if } \mathsf{eval}_v(P_{cond})[i] = \texttt{T} \\ \mathsf{eval}_v(P_{else})[i] & \text{if } \mathsf{eval}_v(P_{cond})[i] = \texttt{F} \\ \mathsf{err} & \text{otherwise} \end{cases}$$

A *(sub)goal* is a vector whose values range over program values $\mathcal{V}$ and a distinguished "don't care" value `?`. The *root goal* is a goal consisting of the desired outputs from the given examples, namely $root = \langle out_1, \cdots, out_{|Ex|} \rangle$. For a value vector v and a (sub)goal g, we say that v *matches* g (and write $match(v, g)$) if for every i, either $v[i] = g[i]$ or $g[i] = \texttt{?}$.

The program synthesis problem can be formalized as follows: find a program P such that $\mathsf{eval}_v(P)$ matches *root*.

Escher Formalized. The synthesis procedure of Escher is pictured in Figure 3 as a nondeterministic transition system. Escher takes as input a synthesis task $(Ex, Comps)$ and synthesizes a program that matches the input-output examples given in Ex. In the following, we first give a high-level overview of Escher, and then describe the system more formally.

A configuration of Escher is a triple $\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex} \rangle$ consisting of a set of synthesized programs syn, a goal graph $\mathsf{goalGraph}$, and a list of input-output examples ex. The procedure begins by applying the Init rule, which initializes syn to be the set of input variables *Vars* and $\mathsf{goalGraph}$ to be the goal graph consisting only of a single goal node *root* (representing the desired outputs obtained from the examples *Ex*) and initializes ex to be *Ex*.

$$\frac{}{\langle Vars, (\{\{root\}, \emptyset, \emptyset, root), Ex\rangle} \textsc{Init}$$

$$\frac{c \in Comps \qquad P_1 \in \mathsf{syn} \qquad \cdots \qquad P_{\mathsf{a}(c)} \in \mathsf{syn} \qquad P = c(P_1, \ldots, P_{\mathsf{a}(c)})}{\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex}\rangle \to \langle \mathsf{syn} \cup \{P\}, \mathsf{goalGraph}, \mathsf{ex}\rangle} \textsc{Forward}$$

$$\frac{\begin{array}{c} cond \in \mathbb{B}^{|\mathsf{ex}|} \qquad g \in G \qquad r \text{ is fresh} \\ bthen = g|cond \qquad belse = g|\neg cond \\ G' = G \cup \{cond, bthen, belse\} \qquad R' = R \cup \{r\} \\ E' = E \cup \{(r,g), (cond, r), (bthen, r), (belse, r)\} \end{array}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \to \langle \mathsf{syn}, (G', R', E', root), \mathsf{ex}\rangle} \textsc{SplitGoal}$$

$$\frac{\begin{array}{c} P_1, P_2, P_3 \in \mathsf{syn} \qquad r \in R \qquad (r,g_1), (r,g_2), (r,g_3) \in E \\ match(\mathsf{eval}_v(P_1), g_1) \qquad match(\mathsf{eval}_v(P_2), g_2) \qquad match(\mathsf{eval}_v(P_3), g_3) \\ P = \texttt{if } P_1 \texttt{ then } P_2 \texttt{ else } P_3 \end{array}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \to \langle \mathsf{syn} \cup \{P\}, (G, R, E, root), \mathsf{ex}\rangle} \textsc{Resolve}$$

$$\frac{P \in \mathsf{syn} \qquad match(\mathsf{eval}_v(P), root) \qquad \mathsf{ex}{\downarrow_P} \not\subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \to \langle \mathsf{syn}, (\{root'\}, \emptyset, \emptyset, root'), \mathsf{ex} \cup \mathsf{ex}{\downarrow_P}\rangle} \textsc{Saturate}$$

$$\frac{P \in \mathsf{syn} \qquad match(\mathsf{eval}_v(P), root) \qquad \mathsf{ex}{\downarrow_P} \subseteq \mathsf{ex}}{\langle \mathsf{syn}, (G, R, E, root), \mathsf{ex}\rangle \to P} \textsc{Terminate}$$

Fig. 3. Escher synthesis algorithm

The rule Forward implements the forward search part of Escher: a new program is added to syn by applying a component to a vector of programs that have already been synthesized (members of syn). The rules SplitGoal and Resolve implement the conditional inference part of the search by manipulating the goal graph – we will explain these rules further in the following. The Saturate rule adds new input-output examples to ex; if Escher synthesizes a recursive program, we must check that the program also produces the correct results for all the recursive calls in order to ensure that the synthesized program is a solution to the synthesis task. Finally, the Terminate rule terminates the algorithm when a program matching the root goal has been synthesized.

The Goal Graph. We now describe the data structure and technique used for synthesizing conditionals. A goal graph is a bipartite graph $(G, R, E, root)$, where:

- G is a set of value vectors representing *goals* that need to be achieved,
- R is a set of *resolvers* connecting goals to subgoals,
- $E \subseteq G \times R \cup R \times G$ is a set of edges connecting goals and resolvers, and
- $root \in G$ is a distinguished *root goal*.

We assume that each resolver $r \in R$ has a single outgoing edge $(r, g) \in E$, the target of which is called the *parent* of r, and three incoming edges

$(g_1, r), (g_2, r), (g_3, r)$, denoting the *cond*, *then*, and *else* goals that need to be synthesized to synthesize a program solving goal g. We call g_1, g_2, and g_3, *subgoals* of r and g. We also assume that, with the exception of the root goal *root* (which has no outgoing edges), every goal has at least one outgoing edge (i.e., it is a subgoal of at least one resolver).

Example 1. Consider the goal graph in Figure 2. The set of goals $G = \{\langle \texttt{0,1,2}\rangle, \langle \texttt{T,F,F}\rangle, \langle \texttt{0,?,?}\rangle, \langle \texttt{?,1,2}\rangle\}$, the set of resolvers $R = \{\texttt{P}_r\}$, and root of the graph is $\langle \texttt{0,1,2}\rangle$. The graph specifies that in order to synthesize a program for $\langle \texttt{0,1,2}\rangle$, one could synthesize three programs satisfying the three other vectors in G. □

At a high-level, a goal graph goalGraph can be viewed as an AND-OR graph. That is, the goal graph specifies that to synthesize a program satisfying a goal g, then programs satisfying *all* subgoals g_1, g_2, g_3 of *one* of the resolvers r (s.t. $(r, g) \in E$) have to be synthesized.

The rule SplitGoal updates a goal graph by including a new resolver for a given goal. This is accomplished by selecting an arbitrary Boolean vector $cond \in \mathbb{B}^{|\mathsf{ex}|}$ and a goal g. From *cond* and g, we compute a pair of *residual goals* $bthen = g|cond$ and $belse = g|\neg cond$, which agree with g on positions where *cond* is true (or false, in the case of *belse*), and otherwise have don't-care values. Formally,

$$(g|cond)[i] = \begin{cases} g[i] & \text{if } cond[i] \\ \texttt{?} & \text{otherwise} \end{cases} \qquad (g|\neg cond)[i] = \begin{cases} g[i] & \text{if } \neg cond[i] \\ \texttt{?} & \text{otherwise} \end{cases}$$

SplitGoal creates a new resolver for g with three new sub-goals: one for *cond*, one for *bthen*, and one for *belse*, and adds them to the goal graph.

Termination Argument. The procedure described in the preceding is suitable for synthesizing non-recursive programs, but a termination argument is required to synthesize recursive functions. To see why a termination argument is required, consider that (in its absence) the program $\mathsf{self}(x_1, \ldots, x_{\mathsf{a(self)}})$ is always a solution (since this program always matches the *root* goal). This solution should be excluded from the search space because it does not terminate.

We remove non-terminating programs from the search space by redefining the eval_v on the recursive component self so that an error is produced on arguments that are not decreasing, according to the well-founded relation $\prec$ on $\mathcal{V}$. Formally, we define

$$\mathsf{eval}_v(\mathsf{self}(P_1, \ldots, P_{\mathsf{a(self)}}))[i] = \begin{cases} \mathsf{eval}(\mathsf{self}, arg) & \text{if } arg \prec^* in_i \\ \mathsf{err} & \text{otherwise} \end{cases}$$

where $arg = (\mathsf{eval}_v(P_1)[i], \ldots, \mathsf{eval}_v(P_{\mathsf{a(self)}})[i])$ and $\prec^*$ indicates the well-founded relation $\prec$ on $\mathcal{V}$ extended to the lexicographic well-founded relation on $\mathcal{V}^{\mathsf{a(self)}}$.

Example 2. Recall the example from Section 2. To ensure termination of the resulting program for computing list length, Escher enforced that the list with which the recursive call to `length` is made follows the common well-founded

order for the list data type: the length of the list is decreasing. For example, suppose we synthesize the program `length(i)` (where `i` is the only input variable) using FORWARD. Then we may not apply the TERMINATE rule, since $\mathsf{eval}_v(\texttt{length(i)}) = \langle \texttt{err,err,err} \rangle$, which does not match the root goal $\langle \texttt{0,1,2} \rangle$.

Saturation. Finally, we discuss our SATURATE rule. The reason for including this rule is illustrated by the following example:

Example 3. Consider the `length` synthesis task introduced in Section 2, and suppose that our examples are (`[],0`) and (`[1,2],2`), such that *root* is $\langle 0, 2 \rangle$. Let P be the program

```
if isEmpty(i) then 0
else if isEmpty(tail(i)) then 0
     else inc(length(tail(i)))
```

Then $match(P, root)$, but P is not a solution to the synthesis task. □

The problem with the above example is that eval_v uses the *Oracle* to evaluate recursive calls rather than the synthesized program. This is required because ESCHER constructs programs in a bottom-up fashion, so eval_v must be able to evaluate self before a candidate solution is fully constructed. So, on the above example, P returns `1` result on input `[1,2]`, but $match(P, root)$ holds because it uses the *Oracle* to evaluate `length(tail(i))`. The SATURATE rule (and the $\mathsf{ex}{\downarrow_P} \subseteq \mathsf{ex}$ side-condition of TERMINATE) resolves this problem, as we will describe in the following.

We define $\mathsf{ex}{\downarrow_P}$ to be the set of all input-output examples (I, O) such that there is some recursive call in P that evaluates $\mathsf{self}(I)$ for one of the input-output examples in ex. On our example, $\mathsf{ex}{\downarrow_P} = \{(\texttt{[2],1})\}$. So SATURATE adds this new example to ex. For completeness, we formally define $\mathsf{ex}{\downarrow_P}$ in [4].

Intuitively, the $\mathsf{ex}{\downarrow_P}$ is the set of input-output examples upon which the examples in ex depend. A *saturated* set of examples (one in which $\mathsf{ex}{\downarrow_P} \subseteq \mathsf{ex}$) does not depend on anything – so if a program P is correct on a saturated example set, then it is guaranteed to be correct for all the examples. The SATURATE rule simply adds such "dependent examples" to the set of examples for which we are obligated to prove correctness.

Completeness. We conclude this section with a statement of the completeness of our synthesis algorithm. ESCHER is complete in the sense that if there exists a solution to the synthesis task within the search space, it will eventually find one. Theorem 1 states this property formally.

Theorem 1 (Relative Completeness). *Given a synthesis task* $(Ex, Comps)$*, suppose there exists a program* $P \in Program$ *such that* $match(\mathsf{eval}_v(P), goal)$*,*[1] *where* $goal = \langle out_1, \ldots, out_{|Ex|} \rangle$ *Then for all reachable configurations* $\langle \mathsf{syn}, \mathsf{goalGraph}, \mathsf{ex} \rangle$ *of* ESCHER*, there exists a run of the algorithm ending in a solution to the synthesis problem.*

[1] Note that this condition implies that P terminates according to the argument above.

Assuming a natural fairness condition on sequences of ESCHER rule applications, we have an even stronger result: if a solution to the synthesis task exists, ESCHER will find one.

3.3 Search Guidance

In this section, we discuss techniques we use to guide the search procedure used in ESCHER. These techniques are essential for turning the naïve transition system presented in Figure 3 into a practical synthesis algorithm.

Heuristic Search. In a practical implementation of ESCHER, we require a method for choosing which rule to apply in any given configuration. In particular, FORWARD has a large branching factor, so it is necessary to determine which component to apply to which subprograms at every step.

Our method is based on using a *heuristic function* $h : \mathit{Program} \to \mathbb{N}$ that maps programs to natural numbers. The lower the heuristic value of a program, the more it is desired.

A simple example of a heuristic function is the function mapping each program to its size. When ESCHER is instantiated with this heuristic function, the search is biased towards smaller (more desirable) programs. The design of more sophisticated heuristic functions for program synthesis is an important and interesting problem, but is out of the scope of this paper. A promising approach based on machine learning is presented in [18].

Observational Equivalence Reduction. The synthesis problem solved by ESCHER requires a program to be synthesized that matches a given list of input-output examples. Programs that evaluate to the same outputs for the inputs given in ex are indistinguishable from the perspective of this task. This idea yields a technique for reducing the size of search space, which we call observational equivalence reduction.

We define an equivalence relation $\equiv$ on programs such that $P \equiv Q$ iff $\mathsf{eval}_v(P) = \mathsf{eval}_v(Q)$. Whenever a new program P is synthesized, ESCHER checks whether a program Q has already been synthesized such that $P \equiv Q$: if such a program Q exists, the program P is discarded. This ensures that at most one representative from each equivalence class of $\equiv$ is synthesized. The correctness of observational equivalence reduction is implied by the following proposition.

Proposition 1. *Let P, Q, Q' be programs such that $Q \equiv Q'$, and that P is a solution to the synthesis task (i.e., $\mathit{match}(\mathsf{eval}_v(P), \mathit{root})$). Let P' be the program obtained from P by replacing every instance of Q with Q'. Then P' is also a solution to the synthesis task.*

A corollary of this proposition is that our completeness theorem (Theorem 1) still holds in the presence of observational equivalence reduction.

Rule Scheduling. We now briefly comment on some practical considerations involved in scheduling the rules presented in Figure 3.

```
let hbal_tree n =
  if leq0(div2(n)) then createLeaf(0)
  else createNode(0, hbal_tree(div2(dec(n))), hbal_tree(div2(n)))

let stutter l =
  if isEmpty(x) then emptyList
  else cons(head(x), cons(head(x), stutter(tail(x))))
```

Fig. 4. Output by ESCHER for height balanced binary tree `hbal_tree`, assuming `n > 0`, and `stutter`

In our implementation of ESCHER, the FORWARD rule is applied in a dynamic programming fashion, as demonstrated in Section 2. Whenever a new program P is synthesized, we apply SATURATE/TERMINATE to check if P is a solution to the synthesis problem. If not, we apply RESOLVE eagerly to close as many goals as possible. We then apply SPLITGOAL if P matches *some* positions of an open goal. For example, if $\langle$`0,0,2,2`$\rangle$ is an open goal and P computes $\langle$`0,1,2,3`$\rangle$, then we apply SPLITGOAL with the Boolean condition $\langle$`T,F,T,F`$\rangle$ (i.e., for each position i, the condition at position i is `T` if P matches the goal at i and `F` otherwise).

Since the SATURATE burdens the user by requiring them to provide additional input/output examples, it may be desirable to schedule rules so that SATURATE is rarely applied. To accomplish this goal, we may assign high heuristic values to programs which require additional user input to bias the search away from applying the SATURATE rule.

4 Implementation and Evaluation

We have implemented an OCaml prototype of ESCHER in a modular fashion, allowing heuristic functions and components written as OCaml functions to easily be plugged in. Our goal in evaluating ESCHER is as follows: (1) Study the effectiveness of ESCHER on a broad range of problems requiring recursive solutions. (2) Evaluate the performance impact of ESCHER's key goal graph concept and its observational equivalence search guidance heuristic. (3) Evaluate ESCHER against SAT/SMT-based techniques by comparing it to the state-of-the-art tool SKETCH [23].

Benchmarks. Our benchmark suite consists of a number of recursive integer, list, and tree manipulating programs, which were drawn from functional programming assignments, standard list and tree manipulation examples, and classic recursive programming examples. The types of programs we have synthesized with ESCHER include tail recursive, divide-and-conquer, as well as mutually recursive programs, thus demonstrating the flexibility and power of the algorithm in this setting. For example, Figure 4 shows two functions synthesized by ESCHER. The first one, `hbal_tree`, constructs a height-balanced binary tree of a given size `n` using a divide-and-conquer recursive strategy to construct the left

Component Type	Supplied Components
Boolean	`and, or, not, equal, leq0, isEmpty`
Integer	`plus, minus, inc, dec, zero, div2`
List	`tail, head, cat, cons, emptyList`
Tree	`isLeaf, treeVal, treeLeft, treeRight createNode, createLeaf`

Fig. 5. Base set of components used in experiments

#	Program	EscherS	ObsEqOff	GGOff	AllOff	EscherD
	Tree Programs					
1	`collect_leaves`	0.044	0.092	68.928	81.813	0.036
2	`count_leaves`	0.056	0.200	9.345	12.317	0.044
3	`hbal_tree`	1.520	MEM	TIME	MEM	TIME
4	`nodes_at_level`	10.741	MEM	TIME	MEM	0.544
	List Programs					
5	`compress`	0.060	0.520	176.419	MEM	0.440
6	`concat`	0.060	0.144	3.460	5.340	0.284
7	`drop`	0.016	0.044	0.708	0.876	0.024
8	`insert`	2.108	MEM	TIME	MEM	14.993
9	`last`	0.020	0.100	0.292	0.452	0.264
10	`length`	0.008	0.040	0.128	0.220	0.328
11	`reverse`	0.224	10.513	7.792	12.489	0.352
12	`stutter`	0.552	67.332	24.698	42.039	TIME
13	`sum`	0.064	0.432	0.904	1.548	0.204
14	`take`	0.248	7.012	20.925	27.406	1.620
	Integer Programs					
15	`fib`	0.120	1.212	40.091	68.232	TIME
16	`gcd`	0.020	0.016	TIME	MEM	0.024
17-1	`iseven`	0.016	0.020	0.012	0.020	TIME
17-2	`isodd`	0.024	0.040	0.020	0.048	0.056
18	`modulo`	0.044	0.460	0.524	0.752	0.080
19	`mult`	0.108	4.168	8.849	12.905	9.461
20	`square`	0.124	1.308	6.296	13.141	0.116
21	`sum_under`	0.004	0.012	1.456	2.824	0.088

Fig. 6. Synthesis time of Escher instantiations using the 22 components indicated in Figure 5. Time is in seconds. TIME denotes a timeout, where the time limit is 5 minutes; MEM denotes that the synthesis process exceeded the 1GB memory limit.

and right subtrees of each node in the tree separately. The second function, `stutter`, duplicates each element in a list. Due to lack of space, we describe our benchmark set in detail in [4].

In order to synthesize these programs, we supplied Escher with a base set of components shown in Figure 5. For all synthesis tasks, this same set of components was used. Our goal with this decision is two-fold: (1) Model a user-friendly environment where the user is not forced to provide a different focused set of components for different tasks, since this requires non-trivial thinking on the part of the user. (2) Demonstrate Escher's ability to synthesize non-trivial programs in the presence of superfluous components.

Our base components cover most basic Boolean, integer, list, and tree operations. For example, Boolean components supply all logical connectives as well as equality checking.

Experimental Setup and Results. We will use EscherS to refer to an instantiation of Escher with the size heuristic for search guidance, and EscherD

to refer to an instantiation with the program depth heuristic (i.e., $h(P)$ is program depth). To study the effects of the goal graph, we implemented a configuration of Escher called GGOff that synthesizes conditionals using the technique employed by [10,13], where an if-then-else component is used to synthesize conditionals and the goal graph is disabled. To study the effects of observational equivalence, we implemented a configuration of Escher called ObsEqOff, where observational equivalence is not checked and all programs are considered. Additionally, AllOff represents a configuration of Escher where both observational equivalence and the goal graph are not used. Except for EscherD, all aforementioned configurations use the size of the program heuristic to guide the search.

We started all configurations of Escher with a minimal number of input-output tuples required to synthesize a correct program for each benchmark (i.e., without having to ask the *Oracle*). Figure 6 shows the number of seconds required by each configuration of Escher to synthesize a correct implementation of the given synthesis task. For example, row 3 shows that EscherS synthesizes the tree manipulating program `hbal_tree` in 2 seconds, whereas ObsEqOff,GGOff, AllOff, as well as EscherD fail to produce a result in the allotted time and memory.

#Comps	`mult`	`modulo`	`sum_under`	`iseven`	`isodd`
0	0.705	0.025	24.339	0.011	0.012
1	12.001	0.069	36.810	0.016	0.015
2	12.570	0.081	42.909	0.018	0.021
3	16.703	0.119	40.952	0.017	0.025
4	16.681	0.188	59.905	0.017	0.028
5	36.269	0.129	66.622	0.020	0.026

Fig. 7. Sketch evaluation

Our results demonstrate the ability of Escher to synthesize non-trivial programs in a very small amount of time, typically less than a second. Moreover, for a large number of programs, not using the goal graph causes the tool to timeout (e.g., `gcd`) or spend a considerable amount of time in synthesis (e.g., `compress`). This demonstrates the power of our technique for synthesizing conditionals in comparison with the naïve method of using an if-then-else component. A similar effect is observed in ObsEqOff, where all synthesized programs are considered. We also observe that on our suite of benchmarks, the program size heuristic outperforms the depth heuristic.

Comparison with Sketch. Sketch [23] is a state-of-the-art synthesis tool that accepts as input a *sketch* (partial program) and a reference implementation (oracle). The sketch is written in a C-like programming language that includes the construct "`??`", denoting an unknown constant value. Sketch then uses SAT-solving to find constants to replace each occurrence of `??` with an integer such that the resulting program is equivalent to a reference implementation. In [10], the authors compare their SMT component-based synthesis technique for straight line programs against Sketch. This is done by encoding the task for searching for a straight line program composed of a set of components as a sketch. For example, given a choice of two components, one can encode the choice as `if (??) then comp1() else comp2()`

To evaluate ESCHER's heuristic search approach against SAT-based techniques, we encoded our component-based synthesis tasks as sketches in a similar fashion to [10], with the addition that a reference specification was also encoded as a component (in order to simulate a recursive call). To ensure termination of the synthesized program, we encoded the same termination argument used by ESCHER (Section 3). Our encoding produces sketches of size linear in the number of components.

There are two limitations to applying SKETCH to our suite of benchmarks: (1) SKETCH can only be applied to the integer benchmarks, since it does not support tree and list data structures; and (2) in order to successfully synthesize programs with SKETCH, we had to supply it with the top level conditional in each benchmark, thus aiding SKETCH by restricting the search space.

On `fib`, `gcd`, and `sum_under`, SKETCH exhausted the allotted 2GB of memory. On `square`, SKETCH returned an error. Figure 7 shows the time taken by SKETCH[2] on the rest of the integer benchmarks where it successfully synthesized a program. The column #COMPS denotes the number of superfluous components provided in the sketch (i.e., components not required for synthesizing the task in question). For example, for `mult`, when the number of components is exactly what is required for synthesis, SKETCH generated a program in 0.71 seconds, but when the number of extra components is 3, SKETCH required 16.7 seconds to synthesize a solution. We observe that the time taken by SKETCH steadily increases in `mult` and `sum_under` as we increase the number of superfluous components. In contrast, ESCHER's results in Figure 6 were obtained by supplying ESCHER with all the 22 components, demonstrating the scalability of ESCHER in the presence of superfluous components.

In summary, our results demonstrate the efficiency of ESCHER at synthesizing a broad range of programs, and emphasize the power of our goal graph data structure at synthesizing conditionals. Moreover, we show that our prototype implementation of ESCHER can outperform a state-of-the-art SAT-based synthesis tool at synthesizing recursive programs from components.

5 Related Work

For a recent survey of various techniques and interaction models for synthesis from examples, we refer the reader to [9].

In version-space algebras, the idea is to design data structures that succinctly represent all expressions/programs that are consistent with a given set of examples. Mitchell [19] pioneered this technique for learning Boolean functions. Lau et al. [15] adapted the concept to *Programming By Demonstration* (PBD) [2], where the synthesizer learns complex functions for text editing tasks. More recently, version-space algebras have been used for data manipulation in spreadsheets, e.g., string transformations [8], number transformations [22], and table transformations [12]. These techniques are limited to domain-specific languages,

[2] Only synthesis time is reported – verification time is not counted.

and different synthesis algorithms are required for different domains. In contrast, ESCHER is parameterized by the components used, thus offering a flexible domain-agnostic synthesis solution.

Explicit search techniques enumerate the space of programs until a program satisfying the given examples is found. This appears in the context of AI planning [20,5], where the search is directed by a goal-distance heuristic. Machine learning techniques have also been used for guiding the search using textual features of examples [3]. These techniques have been mostly limited to synthesizing straight line programs, whereas ESCHER can discover recursive programs.

SAT and SMT solvers have also been used for synthesis. Sketching [23] is the most prominent technique in this category. It accepts a program with holes, and uses a SAT solver to fill the holes with constants to satisfy a given specification (represented as a program). This is performed by bit-blasting the program and encoding it as a formula. In [13], SMT solvers are used to synthesize straight line bit-manipulating programs by interacting with the user via input-output examples. In contrast to these techniques, ESCHER is not restricted by the theories supported by the SMT solver. Also, as we have shown experimentally, sketching is highly sensitive to superfluous components, and even required the top-level conditional in the program to be supplied for successful synthesis. Moreover, ESCHER's heuristic search strategy provides a direct way of adding search preferences/guidance.

The field of inductive logic programming (ILP) was spawned by the work of Shapiro on the Model Inference System (MIS) [21] and by the work of Summers on LISP synthesis [24], among others. Flener and Yilmaz [7] present a nice survey of this rich area. MIS performs synthesis in an interactive manner using search (like ESCHER). However, it scales by fixing errors in a current (incorrect) program. We do not fix incorrect programs but build one from scratch. The idea in Summer's work and its recent incarnation [14] is to start by synthesizing a non-recursive program for the given examples. Then, by looking for syntactic patterns in the synthesized program, the non-recursive program is generalized into a recursive one. ESCHER's approach differs significantly from these techniques, since the whole synthesis algorithm is based on search, and there is no distinction between finding non-recursive programs and generalization. Moreover, ESCHER does not require a "good" set of examples to successfully synthesize a program. Instead, ESCHER can interactively query the user/oracle for more examples (if the initial set does not suffice) until it finds a solution. Summer's line of work was also extended by Flener in his DIALOGS system [6], which is also interactive and features abduction as well (like ESCHER, which abduces conditions of if-then-else statements). However, ESCHER is based on heuristic search to make the process efficient, while DIALOGS uses a non-deterministic algorithm in order to also synthesize alternative programs. DIALOGS can also handle non-ground I/O tuples and can additionally (heuristically) detect the need to invent a help function that is itself recursively defined.

6 Conclusion and Future Work

We have presented ESCHER, a generic and efficient algorithm that interacts with the user via input-output examples, and synthesizes recursive programs implementing intended behaviour. Our work presents a number of interesting questions for future consideration. On the technical side, we would like to extend ESCHER to synthesize loops, alongside recursion. To improve ESCHER's ability to synthesize constants, it would be interesting to combine ESCHER's heuristic search with an SMT-based search. For example, ESCHER can heuristically decide to use an SMT solver to check if there is a solution that uses synthesized constants within n steps for a given input-output example. On the application side, it would be interesting to study the applicability of ESCHER as an *intelligent tutoring system*, where students can learn recursion as a programming paradigm by interacting with the synthesizer, e.g., for suggesting different solutions or providing hints for completing student solutions.

References

1. Flash Fill (Microsoft Excel 2013 feature), `http://research.microsoft.com/users/sumitg/flashfill.html`
2. Your wish is my command: programming by example. Morgan Kaufmann Publishers Inc. (2001)
3. Aditya Menon, S.G.B.L., Tamuz, O., Kalai, A.: A machine learning framework for programming by example. In: ICML 2013 (2013)
4. Albarghouthi, A., Gulwani, S., Kincaid, Z.: Recursive program synthesis, `http://www.cs.toronto.edu/~aws/papers/cav13a.pdf`
5. Bonet, B., Geffner, H.: Planning as heuristic search. Artificial Intelligence 129(1-2), 5–33 (2001)
6. Flener, P.: Inductive logic program synthesis with dialogs. In: Inductive Logic Programming Workshop, pp. 175–198 (1996)
7. Flener, P., Yilmaz, S.: Inductive synthesis of recursive logic programs: Achievements and prospects. J. Log. Program. 41(2-3), 141–195 (1999)
8. Gulwani, S.: Automating string processing in spreadsheets using input-output examples. In: Proc. of POPL 2011, pp. 317–330 (2011)
9. Gulwani, S.: Synthesis from examples: Interaction models and algorithms. In: Proc. of SYNASC (2012) (Invited talk paper)
10. Gulwani, S., Jha, S., Tiwari, A., Venkatesan, R.: Synthesis of loop-free programs. In: Proc. of PLDI 2011, pp. 62–73 (2011)
11. Gulwani, S., Korthikanti, V.A., Tiwari, A.: Synthesizing geometry constructions. In: Proc. of PLDI 2011, pp. 50–61 (2011)
12. Harris, W.R., Gulwani, S.: Spreadsheet table transformations from examples. In: Proc. of PLDI 2011, pp. 317–328 (2011)
13. Jha, S., Gulwani, S., Seshia, S.A., Tiwari, A.: Oracle-guided component-based program synthesis. In: Proc. of ICSE 2010, pp. 215–224 (2010)
14. Kitzelmann, E., Schmid, U.: Inductive synthesis of functional programs: An explanation based generalization approach. JMLR 7, 429–454 (2006)
15. Lau, T., Wolfman, S.A., Domingos, P., Weld, D.S.: Programming by demonstration using version space algebra. JMLR 53(1-2), 111–156 (2003)

16. Manna, Z., Waldinger, R.J.: A deductive approach to program synthesis. ACM TOPLAS 2(1), 90–121 (1980)
17. Manna, Z., Wolper, P.: Synthesis of communicating processes from temporal logic specifications. ACM Trans. Program. Lang. Syst. 6(1), 68–93 (1984)
18. Menon, A., Tamuz, O., Gulwani, S., Lampson, B., Kalai, A.: A machine learning framework for programming by example. In: ICML (to appear, 2013)
19. Mitchell, T.M.: Generalization as search. Artif. Intell. 18(2), 203–226 (1982)
20. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco (2004)
21. Shapiro, E.Y.: Algorithmic Program DeBugging. MIT Press, Cambridge (1983)
22. Singh, R., Gulwani, S.: Synthesizing number transformations from input-output examples. In: Madhusudan, P., Seshia, S.A. (eds.) CAV 2012. LNCS, vol. 7358, pp. 634–651. Springer, Heidelberg (2012)
23. Solar-Lezama, A., Tancau, L., Bodík, R., Seshia, S.A., Saraswat, V.A.: Combinatorial sketching for finite programs. In: Proc. of ASPLOS 2006, pp. 404–415 (2006)
24. Summers, P.D.: A methodology for lisp program construction from examples. J. ACM 24(1), 161–175 (1977)

Efficient Synthesis for Concurrency by Semantics-Preserving Transformations*

Pavol Černý[1], Thomas A. Henzinger[2], Arjun Radhakrishna[2], Leonid Ryzhyk[3], and Thorsten Tarrach[2]

[1] University of Colorado Boulder
[2] IST Austria
[3] NICTA

Abstract. We develop program synthesis techniques that can help programmers fix concurrency-related bugs. We make two new contributions to synthesis for concurrency, the first improving the efficiency of the synthesized code, and the second improving the efficiency of the synthesis procedure itself. The first contribution is to have the synthesis procedure explore a variety of (sequential) *semantics-preserving program transformations*. Classically, only one such transformation has been considered, namely, the insertion of synchronization primitives (such as locks). Based on common manual bug-fixing techniques used by Linux device-driver developers, we explore additional, more efficient transformations, such as the reordering of independent instructions. The second contribution is to speed up the counterexample-guided removal of concurrency bugs within the synthesis procedure by considering *partial-order traces* (instead of linear traces) as counterexamples. A partial-order error trace represents a set of linear (interleaved) traces of a concurrent program all of which lead to the same error. By eliminating a partial-order error trace, we eliminate in a single iteration of the synthesis procedure all linearizations of the partial-order trace. We evaluated our techniques on several simplified examples of real concurrency bugs that occurred in Linux device drivers.

1 Introduction

We develop program synthesis techniques that can help programmers fix concurrency-related bugs. We place ourselves into a setting all the threads of the program are sequentially correct, i.e., all the errors are due to concurrency. In this setting, our goal is to automatically fix concurrency errors. In other words, the programmer needs to worry only about sequential correctness, and the synthesis tool automatically makes the program safe for concurrent execution.

Our first contribution is to have the synthesis procedure explore a variety of (sequential) *semantics-preserving program transformations* to obtain efficient concurrent code. In existing work on partial program synthesis, mostly only one

* This work was supported in part by the Austrian Science Fund NFN RiSE (Rigorous Systems Engineering), by the ERC Advanced Grant QUAREM (Quantitative Reactive Modeling), and by a gift from Intel Corporation.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 951–967, 2013.

such transformation has been considered: the insertion of synchronization primitives such as locks [15,3]. Our study of real-world concurrency bugs from device drivers shows that only 17% of bugs are fixed using locks. For the remaining bugs, developers use other program transformations that avoid the use of synchronization primitives yielding more efficient code. In particular, the most common fix used for 28% of driver concurrency bugs is *instruction reordering*, i.e., a rearrangement of program instructions that changes the driver's concurrent behavior while preserving the sequential semantics of each thread. For example, a pointer initialization may be moved before an instruction releasing another thread that dereferences the pointer. We develop a technique for automating this type of transformation. We also consider other semantics-preserving transformations inspired by practical bug-fixing techniques. For example, the synthesis tool may repeat idempotent instructions multiple times (we give an example where duplicating an instruction removes a concurrency bug).

Our second contribution is to increase the efficiency of the synthesis procedure itself by considering *partial-order traces* (as opposed to linear traces) as counterexamples in the context of counterexample-guided synthesis. A partial order on the instructions involved in the counterexample represents a set of linear counterexample traces that all lead to the same error. We first find a linear counterexample trace using an off-the-shelf tool, and then generalize it to a partial-order trace. We achieve this generalization by combining ideas from Lipton reduction [9] and error invariants [6]. We relax the ordering of a pair of instructions in the linear trace if swapping these instructions preserves error invariants (and thus the bug can still be reached). Intuitively, the resulting partial-order trace captures the 'true cause' of the bug. For instance, if the linear counterexample includes context switches that are not necessary to reach the bug, these context switches will not be required by the partial-order trace.

A key insight in our algorithm is that given the partial-order trace μ, the problem of eliminating μ can be phrased as the problem of creating a minimal cycle in a graph (representing the partial order) by adding new edges. A graph with a cycle does not allow linearization and hence a cycle corresponds to a set of transformations that together eliminate μ. The additional edges correspond to possible instruction reordering or the insertion of atomic sections. Each additional edge is labeled by a cost (for instance, the length of the atomic section).

We implemented our techniques in a prototype tool called ConcurrencySwapper. As specifications, we handle assertions, deadlocks, and generic conditions such as pointer use before initialization. However, our techniques apply to a larger class of reachability properties. For finding buggy traces, we use the model checker Poirot [1]. If Poirot produces a buggy trace, ConcurrencySwapper generalizes it to a partial-order trace, which it then tries to eliminate first by instruction reordering, and failing that, using an atomic section. Otherwise, the current version of the driver is returned, with all the discovered bugs fixed.

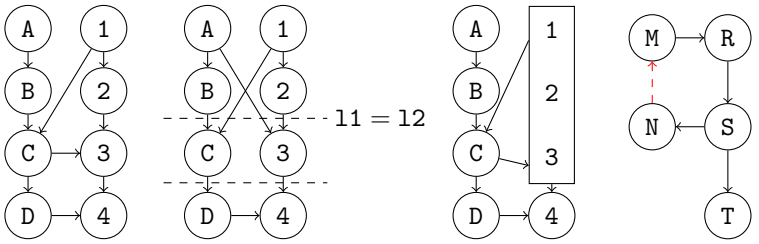
We evaluated our tool on (a) five microbenchmarks that are simplified versions of bugs from Linux device drivers, and (b) a simplified driver for the Realtek 8169 Ethernet controller. The latter had 364 LOC, seven threads, and contained

```
init: x = 0; t1 = F
thread1      thread2
A: l1 = x    1: l2 = x
B: l1++      2: l2++
C: x = l1    3: x = l2
D: t1 = T    4: assert(!t1∨x=2)
```

(a) Concurrent increment

```
init: IntrMask = 0; ready = 0; handled = 0;
init_thread        intr_thread
M: IntrMask = 1    R: assume(IntrMask = 1)
N: ready = 1       S: handled = ready
                   T: assert(handled)
```

(b) Interrupt handling



(c) $\preceq_\pi^1$ (d) $\preceq_\pi^2$ (e) $\preceq_\pi^{1+}$ (f) $\preceq_\theta$

Fig. 1. Illustrative examples

five bugs. In the experiments, we found that: (a) bug finding and verification (in Poirot) dominates time spent generalizing counterexamples, and (b) using generalized counterexamples reduces the number of bug-finding iterations.

Related Work. Synthesis for concurrent programs has attracted considerable research [13,15,3], which is mainly concerned with the insertion of locks. In contrast, we consider general semantics-preserving transformations, a key one being instruction reordering. In [13] and [14], an order for a given set of instructions is synthesized, whereas we are given a buggy program, and we reorder instructions to remove the bug (while preserving the sequential semantics). The main difference from previous work is the algorithm. We generalize counterexample traces to partial-order traces and eliminate them by adding constraints on the instruction order. In contrast, e.g., in [13] the problem of choosing orderings is reduced to resolution of nondeterminism.

Concurrent trace theory has long studied the idea of treating traces as partial orders over events, as in the seminal work on Mazurkiewicz traces (see, for example, [10]). However, their use in counterexample-guided approaches to abstraction refinement, verification, or synthesis has not been considered before. We augment concurrent traces with error invariants, on which there has been recent work in the sequential setting [6].

2 Illustrating Examples

Generalizing Buggy Traces. In Figure 1a, `thread1` and `thread2` concurrently increment `x`. The assertion states that `x` is 2 in the end. It fails in trace $\pi \equiv \mathtt{A} \to \mathtt{B} \to 1 \to \mathtt{C} \to 2 \to \mathtt{D} \to 3 \to 4$, where both threads read the initial `x` value 0, and then write back 1 to `x`. However, π is just one trace exhibiting this bug. For example, swapping `B` and `1` in π gives another buggy trace. Let $\preceq_\pi$ be a

total order where $\texttt{X} \preceq_\pi \texttt{Y}$ iff statement X occurs before Y in π. We relax $\preceq_\pi$ by removing all constraints $\texttt{X} \preceq_\pi \texttt{Y}$ where `X; Y` has the same effect as `Y; X`. This gives us the partial order $\preceq^1_\pi$ (shown in Figure 1c). All traces where the execution order respects $\preceq^1_\pi$ fail the assertion.

For `C` and `3`, the sequence `C; 3` is not equivalent to `3; C` when $\texttt{l1} \neq \texttt{l2}$. However, in all traces of $\preceq^1_\pi$, it can be seen that $\texttt{l1} = \texttt{l2} = 1$, and further, this is sufficient to trigger the bug. These sufficient conditions to trigger bugs are *error invariants*. Using this information, we can further relax $\preceq_\pi$ to $\preceq^1_\pi$ shown in Figure 1d, where the only constraints are that both threads read `x` before either writes to it, and that `D` occurs before `4`. A main component of our synthesis algorithm is the generalization of buggy traces to determine their root cause.

Atomic Sections. We attempt to eliminate the bug represented by $\preceq_\pi$ by adding atomic sections. For example, adding an atomic section around `1`, `2`, and `3` in $\preceq^1_\pi$ gives us $\preceq^{1+}_\pi$ from Figure 1e, where the atomic section is collapsed into a single node. Note that $\preceq^{1+}_\pi$ is not a valid partial order, as there is a cycle of nodes `[1;2;3]` and `C`. Intuitively, the cycle implies that `[1;2;3]` happens both before and after `C`, which is impossible. Hence, adding an atomic section around `[1;2;3]` eliminates all traces represented by $\preceq^1_\pi$ from the program. The atomic section `[1;2;3]` does not eliminate the buggy trace $\texttt{A} \to [1;2;3] \to \texttt{B} \to \texttt{C} \to \texttt{D} \to \texttt{4}$. Analyzing this trace similarly, we find that another atomic section `[A;B;C]` is needed to obtain a correct program.

The number of bug fixing iterations can be reduced using error invariants. For example, in $\preceq^2_\pi$, the atomic section `[1;2;3]` is not sufficient to create a cycle; instead, we immediately see that both `[1;2;3]` and `[A;B;C]` are needed.

Instruction Reordering. The example in Figure 1b is inspired by a real bug from a Linux device driver. Thread `intr_thread` runs when interrupts are enabled, i.e., `IntrMask` is 1, and attempts to handle them; it fails if the driver is not ready. The `init_thread` enables interrupts and readies the driver.

The bug is that interrupts are enabled before the driver is ready, for example, in trace $\theta \equiv \texttt{M} \to \texttt{R} \to \texttt{S} \to \texttt{N} \to \texttt{T}$. Note that statements `M` and `N` are independent, i.e., `M; N` is equivalent to `N; M`. We construct a partial order from θ as before, but remove the constraint $\texttt{M} \preceq_\theta \texttt{N}$, giving us Figure 1f (excluding the dashed edge). Adding the edge $\texttt{N} \to \texttt{M}$ creates a cycle and eliminates the bug. This edge changes the order of `M` and `N`, forcing the order `N; M`. This results in a correct program with the driver ready to handle interrupts before they are enabled.

Following the ideas presented in this section, our synthesis algorithm works by generalizing linear counterexample traces to partial-order traces and eliminating them using atomic section insertion or instruction reordering.

3 Model and Problem Statement

Let V be a set of variables ranging over a domain $\mathcal{D}$. A *V-valuation* is a function $\mathcal{V} : V \to \mathcal{D}$. A *state assertion* ϕ is a first-order constraint over valuations of variables. We model an *instruction* as an assertion τ over V and V'. Intuitively,

V and V' represent the values of the variables before and after the execution of the instruction, respectively. For example, $x' = x + y$ represents `x = x + y` in a C-like language. Given a V-valuations $\mathcal{V}$ and a V'-valuation $\mathcal{V}'$, we write $(\mathcal{V}, \mathcal{V}') \models \tau$ to denote the fact that assertion τ holds for values given by $\mathcal{V}$ and $\mathcal{V}'$. Furthermore, we require that the instruction is deterministic, i.e., for every $\mathcal{V}$, there is at most one $\mathcal{V}'$ such that $(\mathcal{V}, \mathcal{V}') \models \tau$.

We model procedures as *control-flow graphs* (CFGs) with locations labeled with instructions. Formally, a *method* is a tuple $\langle V, I, O, S, \Delta, s_\iota, s_f, inst\rangle$ where (a) V is a set of variables and $I \subseteq V$ and $O \subseteq V$ are *input* and *output* variables respectively; (b) S is a finite set of control locations and s_ι and s_f are the initial and final control locations respectively; (c) $\Delta \subseteq S \times S$ is a set of transitions; and (d) *inst* is a function labeling control locations with instructions. The initial values of the input variables and the final values of the output variables are the arguments and the return values of the method call, respectively. We assume that methods are deterministic, and that all the CFGs are reducible (see [11]).

A *concurrent library* $\mathcal{P}$ is a tuple $\langle M_1, \ldots, M_n\rangle$ of methods with mutually disjoint control locations. Let $locs(M_i)$, $vars(M_i)$, $\Delta(M_i)$ be the control locations, variables, and transitions of M_i, respectively. Further, let $globals(\mathcal{P})$ denote the variables shared among methods, and $locs(\mathcal{P}) = \bigcup_i locs(M_i)$ the locations of $\mathcal{P}$.

Modeling language constructs. Programs are encoded as CFGs in a standard way. For example, `if(x == 0)` is a choice between then and else branches prefixed with `assume(x == 0)` and `assume(x != 0)` respectively. We model assertions in M_i using variable err_{M_i} that is set to 1 when an assertion fails. To block execution after an assertion failure, we replace every instruction τ by $err_{M_i} = 0 \wedge \tau$. Atomic sections are modeled using an auxiliary variable that prevents other methods from running when the program is inside an atomic section.

Semantics. The methods of a library can be executed in parallel, by an unbounded number of threads. We assume that each thread executes one method. Let *Tids* be a set of thread identifiers. A *thread state* is a triple $(tid, s, \mathcal{V})$ where $tid \in Tids$, $s \in locs(M_i)$ is a control location and $\mathcal{C}$ is a $(vars(M_i) \setminus globals(\mathcal{P}))$-valuation. A thread state is initial (resp., final) if the control state is initial (resp. final). A *library state* $(\mathcal{G}, \mathcal{T})$ contains a $globals(\mathcal{P})$-valuation $\mathcal{G}$, and a set $\mathcal{T}$ of thread states with unique thread identifiers. State $(\mathcal{G}, \mathcal{T})$ is final for thread *tid* if $(tid, s, \mathcal{V}) \in \mathcal{T}$ where s is a final control location. We denote by $(\mathcal{G}, \mathcal{T})_{\restriction tid}$ the valuation given by $\mathcal{G} \cup \mathcal{V}$ where $\mathcal{V}$ is such that $(tid, s, \mathcal{V}) \in \mathcal{T}$.

A *single-step execution* of thread *tid* is a triple $((\mathcal{G}, \mathcal{T}), s, (\mathcal{G}', \mathcal{T}'))$ such that there exist M_i, $(tid, s, \mathcal{V}) \in \mathcal{T}$ and $(tid, s', \mathcal{V}') \in \mathcal{T}'$ with (a) $\mathcal{T} \setminus \{(tid, s, \mathcal{V})\} = \mathcal{T}' \setminus \{(tid, s', \mathcal{V}')\}$; and (b) $(\mathcal{G} \cup \mathcal{V}, \mathcal{G}' \cup \mathcal{V}') \models inst(s)$ and $(s, s') \in \Delta(M_i)$. A *trace* π of $\mathcal{L}$ is a sequence $(\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1)s_1 \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ where every thread state in $\mathcal{T}_0$ is initial, and every $((\mathcal{G}_i, \mathcal{T}_i), s_i, (\mathcal{G}_{i+1}, \mathcal{T}_{i+1}))$ is a single-step execution. Since our instructions are deterministic, we write π as $[(\mathcal{G}_0, \mathcal{T}_0)]s_0 \to s_1 \to \ldots$.

A *sequential trace* of $\mathcal{L}$ is a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0 \ldots$ such that all the transitions of a single thread *tid* occur in a contiguous block (say $(\mathcal{G}_k, \mathcal{T}_k)s_k \ldots (\mathcal{G}_{k+l}, \mathcal{T}_{k+l})$) with $(\mathcal{G}_{k+l}, \mathcal{T}_{k+l})$ being final for *tid*. For any such block, we write

$[(\mathcal{G}_k, \mathcal{I}_k) - seq(M_i) \rightarrow (\mathcal{G}_{k+l}, \mathcal{O}_{k+l})]$ where $\mathcal{I}_k$ and $\mathcal{O}_{k+l}$ are the valuation of the input and output variables in $(\mathcal{G}_k, \mathcal{S}_k)_{\uparrow tid}$ and $(\mathcal{G}_{k+l}, \mathcal{S}_{k+l})_{\uparrow tid}$, respectively. The *sequential semantics* $SeqSem(M_i)$ of method M_i is the set of all relations $[(\mathcal{G}, \mathcal{I}) - seq(M_i) \rightarrow (\mathcal{G}', \mathcal{O})]$ that occur in sequential traces of $\mathcal{P}$. Intuitively, the $SeqSem(M_i)$ characterizes all sequential input-output behaviour of a method.

For every trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0 \dots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$, we define $time_\pi : \{0, \dots, n\} \rightarrow \mathbb{N}$ as follows: (a) $time_\pi(0) = 0$; (b) $time_\pi(i+1) = time_\pi(i)$ if s_i is in an atomic section; and (c) $time_\pi(i+1) = time(i) + 1$ otherwise.

The Synthesis Problem. A trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \dots$ of $\mathcal{P}$ is *erroneous* if some library state $(\mathcal{G}_i, \mathcal{T}_i)$ has an err_{M_i} set to 1. Let $\mathcal{P} = \langle M_1, \dots, M_n \rangle$ be a concurrent library. Library $\mathcal{P}$ is *sequentially correct* if every sequential trace of $\mathcal{P}$ is not erroneous. Let M and M' be two methods with the same input and output variables. M and M' are sequentially equivalent, if $SeqSem(M) = SeqSem(M')$.

Let $\mathcal{P}' = \langle M'_1, \dots, M'_n \rangle$ be a concurrent library. We say that $\mathcal{P}'$ is *sequentially equivalent* to $\mathcal{P}$ if, for all i, M_i is sequentially equivalent to M'_i.

The concurrent library synthesis problem asks the following: given a sequentially correct concurrent library $\mathcal{P} = \langle M_1, \dots, M_n \rangle$, output a sequentially equivalent library $\mathcal{P}' = \langle M'_1, \dots, M'_n \rangle$ such that every trace of $\mathcal{P}'$ is not erroneous. Intuitively, the problem asks for a version $\mathcal{P}'$ of $\mathcal{P}$ which is safe for concurrent execution, but has the same sequential semantics. The obvious solution involving adding an atomic section around each method is undesirable. Instead, our approach is as follows: (a) find an erroneous trace π of $\mathcal{P}$; (b) compute a generalization of π; and (c) transform the methods minimally to avoid the bug.

We observe the following about the definition. First, we assume a sequentially correct library to ensure that at least one solution to the synthesis problem exists. However, our approach can be easily extended to not have this assumption; in that case, it may terminate without finding a solution. Second, correctness was specified by assertions in the code. However, our approach works for a more general class of specifications, namely, for safety properties on the global state.

4 Semantics-Preserving Transformations

We present a few sequential-semantics preserving transformations, focusing on statement reordering and insertion of atomic sections. We also give a motivating example for idempotent statement duplication, but do not treat it formally, in the interest of space. Our synthesis algorithm operates by collecting constraints on possible solutions, and hence, we present representations of constraints on possible solutions obtained by the considered transformations.

Statement Reordering. Statement reordering is a transformation that changes the order of statements within a method. Notably, this transformation can change concurrent behavior without changing sequential semantics (e.g., in Figure 1b). A *block* is a single-entry, maximal sequence of control locations $s_0 \dots s_k$ representing straight-line code. For simplicity, we only consider reorderings within blocks. Our techniques can be extended to allow reorderings across block boundaries.

We represent multiple reorderings of method M compactly using a *reordering constraint* $\sqsubseteq \subseteq locs(M) \times locs(M)$, that specifies a partial order on $locs(M)$. We may have $s \sqsubseteq s'$ only if s and s' are from the same block. We write $s \sqsubsetneq s'$, if $s \sqsubseteq s'$ and s is different from s'. Let M' be a method obtained by statement reordering from method M. Method M'satisfies $\sqsubseteq$ if for all $s \sqsubsetneq s'$, s occurs before s' in the corresponding block. A reordering constraint $\sqsubseteq$ is *weaker* than $\sqsubseteq'$ if $s \sqsubseteq s' \implies s \sqsubseteq' s'$.

As our reorderings need to preserve sequential semantics, we can compute some reordering constraints even before considering concurrent executions. The procedure `SemPreservingOrders` computes a reordering constraint $\sqsubseteq$ as follows. It first constructs the total order $\sqsubseteq$ of control states in each block. Then, it picks s and s' such that $s \sqsubsetneq s'$, and checks if $inst(s)$ and $inst(s')$ commute, i.e., we test using a theorem prover two conditions: (a) $inst(s'); inst(s)$ can execute to completion from each state $inst(s); inst(s')$ can; and (b) they have the same effect. If they commute, then the pair (s, s') is removed from $\sqsubseteq$ and we repeat the process. When not such pair exists, we return $\sqsubseteq$. If `SemPreservingOrders` returns $\sqsubseteq$ on input M, then every M' satisfying $\sqsubseteq$ (and obtained by reordering) is sequentially equivalent to M, and no weaker $\sqsubseteq'$ has the same property.

Example 1. Running `SemPreservingOrders` on the code fragment from Figure 1b gives us a single constraint $\texttt{S} \sqsubseteq \texttt{T}$ as all other pairs of statements are independent of each other. In Figure 1a, we get $\texttt{A} \sqsubseteq \texttt{B} \sqsubseteq \texttt{C}$ and $\texttt{1} \sqsubseteq \texttt{2} \sqsubseteq \texttt{3} \sqsubseteq \texttt{4}$.

Atomic Sections. Our second semantics-preserving transformation is insertion of an atomic section. An atomic section encapsulates a set of statements, and ensures that no concurrent thread can interrupt the execution of these statements.

We represent an atomic section by the set of control locations A it contains. An *atomicity constraint* $\alpha \subseteq 2^{locs(\mathcal{P})}$ is satisfied by a set of atomic sections $\{A_1, A_2, \ldots, A_n\}$ if $\forall A \in \alpha. \exists A_i : A \subseteq A_i$. An atomicity constraint α is weaker than an atomicity constraint α' if $\forall A \in \alpha. \exists A' \in \alpha' : A \subseteq A'$. Any set of atomic sections that satisfies α' also satisfies the weaker α.

Combining Atomicity and Reordering Constraints. A *constraint* is an atomicity- and reordering-constraint pair. Constraint $(\alpha, \sqsubseteq)$ is *weaker* than $(\alpha', \sqsubseteq')$ if either α is weaker than α', or $\alpha = \alpha'$ and $\sqsubseteq$ is weaker than $\sqsubseteq'$. Intuitively, we prefer reordering to inserting atomic sections. We define the conjunction of constraints $(\alpha, \sqsubseteq) \wedge (\alpha', \sqsubseteq')$ as $(\alpha'', \sqsubseteq'')$ where:

- $\alpha'' = \alpha \cup \alpha' \cup \{\{s, s'\} \mid ((s \sqsubseteq s' \wedge s' \sqsubseteq' s) \vee (s \sqsubseteq' s' \wedge s' \sqsubseteq s)) \wedge s \neq s'\}$;
- $\sqsubseteq'' = (\sqsubseteq \cup \sqsubseteq') \setminus \{(s, s') \mid ((s \sqsubseteq s' \wedge s' \sqsubseteq' s) \vee (s \sqsubseteq' s' \wedge s' \sqsubseteq s)) \wedge s \neq s'\}$.

Intuitively, if $\sqsubseteq$ and $\sqsubseteq'$ disagree on the order of s and s', we put them in an atomic section. We define $\top$ as a trivial constraint satisfied by all libraries.

Other Transformations. We motivate another sequential-semantics preserving transformation with an example. Some further transformations are in Section 7.1.

Example 2. In Figure 2, the `timer` thread is invoked when `timer_enabled = 1` to handle requests. The device shutdown thread, `shutdown`, handles the remaining requests and disables the timer. There are two correctness conditions: (1) the timer is disabled after device shutdown; and (2) the `unsafe()` function can accessed only by one thread at a time. Condition (2) is violated as statements `1` and `C` can cause `unsafe` to be executed simultaneously. This happens if `C` calls `unsafe`, and after executing a few instructions of `unsafe`, thread `timer` executes and, in the atomic section, calls `unsafe`. One fix is to move `2` before `1`. This introduces a trace where the assertion fails as the timer gets re-enabled by `1` (switching `1` and `2` is not semantics preserving). A possible fix is to execute statement `2` twice, before and after statement `1`.

```
init: timer_enabled=1, halted=0
timer                          shutdown                      work_queue() {
atomic{                        1: work_queue()                 P: unsafe()
 A: assume(timer_enabled)      2: timer_enabled=0              Q: timer_enabled=1
 B: timer_enabled=0            3: assert(!timer_enabled)     }
 C: work_queue()
}
```

Fig. 2. Example for copying idempotent statements

The above example illustrates another useful semantics-preserving transformation, namely, replication of idempotent statements. A statement s occurring after s' can be replicated before s' if `s; s'; s` has the same effect as `s'; s`.

5 Generalizing Counterexamples to Partial-Order Traces

Partial-order traces. A *po-trace* μ of a concurrent library $\mathcal{P}$ is a tuple $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$ where (a) X is a finite set (b) $\preceq$ is a partial-order on X; (c) $loc : X \to Tids \times locs(\mathcal{P})$ is a function labeling X with thread identifiers and control states; and (d) φ_{in} and φ_{end} are state assertions.

A po-trace represents a set of traces of the library. We say a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ is contained in μ if there is a bijection $f : X \to \{0, 1, \ldots, n-1\}$ such that: (a) $f(x) = i \implies loc(x) = (tid, s_i)$ and $\exists \mathcal{V} : (tid, s_i, \mathcal{V}) \in \mathcal{T}_i$; (b) $x_1 \preceq x_2 \implies time_\pi(f(x_1)) \leq time_\pi(f(x_2))$; and (c) $f(x) = 0 \implies (\mathcal{G}_i, \mathcal{T}_i)_{\restriction tid} \models \varphi_{in}) \wedge f(x) = n-1 \implies (\mathcal{G}_n, \mathcal{T}_n)_{\restriction tid} \models \varphi_{end})$. Intuitively, $\pi \in \mu$ if the execution order of statements in π respects the partial order given by $\preceq$, the condition φ_{in} holds at the beginning, and the condition φ_{end} holds at the end. If the order $\preceq$ is linear, we call μ a *linear* po-trace.

As an example, we show how an erroneous trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ such that $\mathcal{G}_n(err) = 1$ is converted into a linear po-trace $\mu_\pi = \langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. The elements of the tuple are defined as follows. (a) X is {0, 1, ..., n-1 }; (b) $\preceq_\pi$ is the natural order on $\mathbb{N}$; (c) $loc(i) = (tid, s_i)$ if $((\mathcal{G}_i, \mathcal{T}_i), s_i, (\mathcal{G}_{i+1}, \mathcal{T}_{i+1}))$ is a single-step execution of thread tid; and (d) φ_{in} expresses that we start from any initial state; (e) φ_{end} expresses that an error is reached (i.e., it is $err = 1$).

Given two po-traces $\mu = \langle \varphi_{in}, X, \preceq_\mu, loc, \varphi_{end} \rangle$ and $\mu' = \langle \varphi_{in}, X, \preceq_{\mu'}, loc, \varphi_{end} \rangle$ sharing the same trace locations X and loc, and assertions φ_{in} and φ_{end}, we say that μ' is a *relaxation* of μ if $\preceq_\mu \supseteq \preceq_{\mu'}$. Intuitively, a relaxed po-trace puts fewer constraints on the order of execution of statements.

Error invariants. Error invariants were introduced in [6] in a sequential setting. Here we use them to generalize counterexamples to partial-order traces. Let μ be a linear po-trace $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. Without loss of generality, let X be $\{0, 1, \ldots, n-1\}$ and let $\preceq$ be the natural order on $\mathbb{N}$. An *error invariant* *ErrInv* is a function from X to state assertions, such that : (a) $ErrInv(0) = \varphi_{in}$ (b) $ErrInv(i)$ (for $0 < i \leq n-1$) over-approximates the set of states reachable at i along μ. That is, if for a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ we have that if φ_{in} holds for $(\mathcal{G}_0, \mathcal{T}_0)$, then $ErrInv(i)$ holds for $(\mathcal{G}_i, \mathcal{T}_i)$. (c) $ErrInv(i)$ under-approximates the set of states from which we can reach the φ_{end} along μ starting from i. That is, if for a trace $\pi = (\mathcal{G}_0, \mathcal{T}_0)s_0(\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1}(\mathcal{G}_n, \mathcal{T}_n)$ we have that if $ErrInv(i)$ holds for $(\mathcal{G}_i, \mathcal{T}_i)$, then φ_{end} holds for $(\mathcal{G}_n, \mathcal{T}_n)$.

As an example, let us consider linear po-trace given by a sequence of statements `A: x=x+1; B:x=2*x; C:y=y+1`, and where φ_{in} is $x = 0 \wedge y = 0$, and φ_{end} is $x > 0 \wedge y > 0$. An error invariant *ErrInv* can be $x > 0 \wedge y \geq 0$ before the execution of `B`, and the same formula before `C`.

We generalize the notion of error invariant to (non-linear) po-traces. Let a po-trace μ be a tuple $\langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$. An error invariant for μ is a function *ErrInv* from X to state assertions such that *ErrInv* is an error invariant for every linear po-trace μ' such that μ is a relaxation of μ'.

5.1 Generalizing Counterexample Traces

We say that a po-trace μ is a counterexample if every trace π contained in μ is erroneous. Given an erroneous trace π, or equivalently, a linear po-trace μ_l, we now present techniques for generalizing it into a non-linear po-trace μ that is a counterexample.

The trace generalization technique proceeds iteratively. Given a po-trace $\mu = \langle \varphi_{in}, X, \preceq, loc, \varphi_{end} \rangle$, in each step, we attempt to relax μ by removing the relation $A \preceq B$ for some $A, B \in X$ where A and B correspond to statements from different threads. Further, we require that $\neg \exists P : A \preceq P \preceq B$. However, we need to ensure that the resultant po-trace remains a counterexample after the relaxation, i.e., that every trace contained in it is an erroneous trace. We formalize this condition below.

Let $C, D \in X$ be such that $C \preceq A \preceq B \preceq D$ and $\forall E \in X : E \preceq C \vee C \preceq E \preceq D \vee D \preceq E$. Further, let $\kappa \subseteq X$ be the set $\{E | C \preceq E \preceq D\} \setminus \{D\}$, i.e., κ represents the set of instructions occur between C and D. We call the triple (C, D, κ) a *border set* of A, B, and $\preceq$.

Let J_C and J_D be the error invariants at C and D. Intuitively, we check that we can get from J_C to J_D for every ordering of instructions in κ allowed by $\preceq \setminus (A, B)$. Formally, let X_1, X_2, $\ldots X_n$ be such that each $X_i \in \kappa$ and $\forall E \in \kappa . \exists i : X_i = E$, and $\forall i : X_i \preceq X_{i+1} \vee X_i = B \wedge X_{i+1} = A$. Let s_i be

the instruction corresponding to X_i. We allow relaxing the condition $A \preceq B$ in a step if and only if the following holds: for every sequence $X_1, X_2, \ldots, X_n$ satisfying the above conditions, the Hoare-triple $\{J_C\}s_1; s_2; \ldots; s_n\{J_D\}$ is valid.

Therefore, the full technique for generalizing a trace is as follows: in each step, we pick A and B, and then check the above conditions. If they hold, we relax by removing the pair (A, B) from $\preceq$. Although this technique is sound and complete for generalizing traces, it can be inefficient due to the large number of complex checks needed in each iteration. Instead, we present an alternative algorithm (Algorithm 1) which is sound, but incomplete. The outline of the algorithm is the same as the complete technique presented above, i.e., in each iteration, the algorithm attempts to relax $A \preceq B$. However, we use two alternative checks.

Note that when we try to relax an edge from A to B, we need to check whether this does not invalidate any previous relaxation. Therefore we recheck all the previously relaxed edges in the border set given by A and B.

Algorithm 1. Generalizing linear counterexamples

Input: linear counterexample po-trace μ_l, error invariant *ErrInv*
Output: counterexample po-trace μ that is a relaxation of μ_l

```
1: μ = μ_l
2: for all A ⪯_{μ_l} B do
3:    removeEdge(A, B, μ)
4:    C, D, κ ← borderSet(A, B, μ)
5:    res ← true
6:    for all U, V ∈ κ do
7:       if U ⋠_μ V ∧ V ⋠_μ U then
8:          res ← res ∧ (check1(U, V, C, D, μ, ErrInv) ∨
                         check2(U, V, C, D, μ, ErrInv))
9:    if ¬ res then addEdge(A, B, μ)
10: return μ
```

Rule 1, implemented in procedure *check1*, allows relaxing the order between statements that commute under certain conditions. Let s_U and s_V be the instructions corresponding to U to V. To relax the edge from U to V, we check if there exists K_1 such that $\{J_C\}s_C\{K_1\}$ is a valid Hoare-triple and $K_1 \wedge s_V; s_U \implies K_1 \wedge s_U; s_V$. Intuitively, we are checking if the instructions s_U and s_V commute given the pre-condition K_1. Further, we require that other instructions do not interfere with K_1, i.e., for all $E \in \kappa$ with instruction s_E, K_1 is preserved under s_E, i.e., $\{K_1\}s_E\{K_1\}$ is a valid Hoare-triple.

Rule 2, implemented in procedure *check2*, allows relaxing the order between statements which do not commute, but ensure the similar post-conditions in both orders. The procedure `check2`(U, V, C, D, μ) works as follows. Let J_C be the error invariant at C, and let J_D be the error invariant at D. Let s_U and s_V be the two instructions at nodes U and V. The procedure returns `true` if and only if there exists two state assertions K_1 and K_2 such that all nodes the following conditions hold: (a) $\{J_C\}s_C\{K_1\}$, $\{K_1\}s_U; s_V\{K_2\}$, and $\{K_1\}s_U; s_V\{K_2\}$ are valid Hoare-triples; and (b) $K_2 \rightarrow J_D$. These conditions state that the error

invariants are sufficient to prove that s_u and s_v commute. Furthermore, let E be any other node in κ, and let s_E be the corresponding instruction. We require that s_E preserves K_1 and K_2, i.e., the following two Hoare-triples are valid: (c) $\{K_1\}s_E\{K_1\}$ (d) $\{K_2\}s_E\{K_2\}$ Intuitively, instead of checking all allowed paths from C to D, we find state assertions K_1 and K_2 that are strong enough to prove commutativity, but are preserved by other statements in κ.

Example 3.
- Consider methods `1: x = 0; 2: x = x++` and `A: x = x++; B: assert(x ≤ 1)`. Here, `1` → `A` → `2` → `B` is an erroneous trace. However, the ordering of `A` and `2` is irrelevant to the bug. This order can be eliminated by applying Rule 1 with precondition $K_1 \equiv$ `true`, as we have `A;2` $\Longrightarrow$ `2;A`.
- Using Rule 1 in the illustrative example (Figure 1a) taking K_1 to be $l_1 = 1 \wedge l_2 = 1$ lets us commute the statements `x = l1` and `x = l2`.
- Consider two methods each with the code `1: x = 3` and `A: x = 2; B: assert(x == 0)`. The erroneous trace here is `1` → `A` → `B`. Here, it is clear that `1` and `A` do not commute, i.e., `1;A` $\not\equiv$ `A;1`. However, in the context of this trace, interchanging `A` and `1` still preserves the error. Therefore, using Rule 2 with $K_1 \equiv$ `true` and $K_2 \equiv x > 0$ relax the ordering between `A` and `1`.

We note that Rule 1 and Rule 2 provide only a sound, not a complete proof system for trace generalization. Application of both these rules involve finding suitable K_1 and K_2. The set of conditions imposed on K_1 and K_2 can be expressed as Horn clauses. Solving Horn clauses (in logics useful for program analysis) is a focus of recent research. Non-recursive version was solved by [7], and recursive Horn clauses are solved successfully using heuristics, for example, in [8]. These techniques can be used to implement `check1` and `check2`.

Theorem 1. *Let μ_l be a linear counterexample po-trace corresponding to an erroneous trace, and ErrInv an error invariant for μ_l. If Algorithm 1 returns po-trace μ on μ_l and ErrInv, then μ is a counterexample and a relaxation of μ_l.*

6 Synthesis by Elimination of Partial-Order Counterexamples

We now present Algorithm 2 to solve the synthesis problem stated in Section 3. It works by finding a buggy trace, generalizing it, and then eliminating it using either an atomic section, or a code reordering. The algorithm maintains an atomicity constraint α and a reordering constraint $\sqsubseteq$. In each iteration, library $\mathcal{P}'$ which satisfies $(\alpha, \sqsubseteq)$ is picked and verified. If correct, it is returned. Otherwise, $(\alpha, \sqsubseteq)$ is strengthened using the generalized counterexample. Note that as `Verify` is solving an undecidable problem, it may not terminate. This results in our algorithm not terminating as well. However, as the constraint is strengthened at each step and only a finite number exist, if all calls to `Verify` terminate, then the algorithm terminates and always returns a correct library. This correct library, in the worst case, will have every method enclosed in an atomic section.

Procedure `SemPreservingOrders` was defined in Section 4. `Generalize` is the Algorithm 1. Procedure `Choose` picks a library satisfying a given constraint. `Eliminate` (see below) finds constraints to eliminate a generalized po-trace.

The basic idea behind generalized trace elimination is that $\preceq_\mu$ encodes the happens-before relation among instructions and hence cannot contain loops. Hence, we aim to enforce minimal constraints to introduce a cycle in the $\preceq_\mu$ relation. We extend the graph representing $\preceq_\mu$ by introducing *constraint edges* corresponding to possible atomic sections and reorderings. We then find the smallest cycles, which correspond to the required minimal constraints.

Algorithm 2. Synthesis algorithm

Input: Library $\mathcal{P}$
Output: Error-free library $\mathcal{P}'$ sequentially equivalent to $\mathcal{P}$

```
1: (α, ⊑) ← (∅, SemPreservingOrders(P))
2: while true do
3:    P' ← Choose(⊑, α)
4:    if Verify(P') return P'
5:    μ ← Generalize(cex(P'), ⊑)
6:    (α, ⊑) ← (α, ⊑) ∧ Eliminate(μ, (α, ⊑))
```

Fix a library $\mathcal{P}$ and a partial-order trace $\mu = \langle \varphi_{in}, X, \preceq_\mu, loc, \varphi_{end} \rangle$ for the remainder of this section. The *elimination graph* $G(\mu, \alpha, \sqsubseteq) = (S, E)$ is a weighted graph with vertices $S = X$. The edges $E \subseteq S \times \mathbb{N} \times S$ are described below. Let $x, x' \in S$ and $loc(x) = (tid, s)$ and $loc(x') = (tid', s')$. The function *cons* assigns a constraint to each edge of the elimination graph. We have $(x, w, x') \in E$ if:

- $tid \neq tid' \wedge x \preceq_\mu x' \wedge w = 1 \wedge \neg\exists x'' : x \preceq_\mu x'' \preceq_\mu x'$. In this case, we define $cons((x, w, x')) = \top$.
- $tid = tid'$, where $x \preceq_\mu x'$ and either x and x' belong to different blocks or $s \sqsubseteq s'$. We have $w = |\{x'' \mid x \preceq_\mu x'' \preceq_\mu x'\}|$. Here, we let $cons((x, w, x')) = \top$. These edges correspond to happens-before relations that hold due to $\sqsubseteq$.
- $tid = tid' \wedge s' \sqsubseteq s$ and $w = A \cdot |\{s'' \mid s' \preceq_\mu s'' \preceq_\mu s\}|$ for some constant $A \in \mathbb{N}$. Here, we define $cons((x, w, x')) = (\{s, s'\}, \emptyset)$. Such edges correspond to adding an atomic section around s and s'. We give the atomic section a cost proportional to the minimum number of control locations it contains.
- $tid = tid' \wedge s \not\sqsubseteq s' \wedge s' \not\sqsubseteq s$ and $w = R \cdot |\{(s'', s''') \mid s'' \not\sqsubseteq s''' \wedge s \sqsubseteq s'' \wedge s''' \sqsubseteq s'\}$ for some $R \in \mathbb{N}$. Here, we define $cons((x, w, x')) = (\emptyset, \{(s, s')\})$. This edge corresponds to forcing the order s before s' and has a cost proportional to the number of additional statement orders the constraint implies.

Intuitively, an edge (x, w, x') with $cons((x, w, x')) = \top$ represents a happens-before relation true in any $\mathcal{P}'$ satisfying $(\alpha, \sqsubseteq)$. Every remaining edge (x, w, x') is a happens-before relation true in any library satisfying $cons((x, w, x'))$. We pick A much larger than R to prefer solutions having only reorderings rather than atomic sections (picking A and R such that $A > R \cdot |X|^2$ is sufficient).

Let $x_0 \ldots x_{n-1} x_0$ be a cycle in the elimination graph for a po-trace μ and $(\alpha, \sqsubseteq)$ such that $loc(x_0) = (tid, s) \wedge loc(x_{n-1}) = (tid', s')$ and $tid \neq tid'$. We call

such a cycle an *elimination cycle*. We show that any elimination cycle gives us a constraint that eliminates all traces in po-trace μ. From the elimination cycle, we obtain the following constraint $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$. This is the constraint returned by `Eliminate` (called from Algorithm 2). Fix constraint $(\alpha, \sqsubseteq)$. A constraint $(\alpha', \sqsubseteq')$ *eliminates* a po-trace μ iff all libraries satisfying $(\alpha, \sqsubseteq) \wedge (\alpha, \sqsubseteq)$ and sequentially equivalent to $\mathcal{P}$ do not share a trace with μ.

Theorem 2. *Let $G(\mu, (\alpha, \sqsubseteq))$ contain an elimination cycle $x_0 x_1 \ldots x_{n-1} x_0$. Then, $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$ eliminates the po-trace μ.*

Proof. Say $\pi = (\mathcal{G}_0, \mathcal{T}_0) s_0 (\mathcal{G}_1, \mathcal{T}_1) \ldots s_{n-1} (\mathcal{G}_n, \mathcal{T}_n) \in \mu$ and let $f : X \to \{1, \ldots, n\}$ be the bijection witnessing the containment. Any trace π in $\mathcal{P}'$ satisfying $(\alpha, \sqsubseteq)$ and $cons(x_i, x_{i+1})$ has $time(f(x_i)) \leq time(f(x_{i+1}))$. Hence, any trace π satisfying $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$ and $(\alpha, \sqsubseteq)$ satisfies $time(f(x_0)) \leq time(f(x_{n-1}))$. However, as (x_{n-1}, x_0) is an edge in the elimination graph where x_0 and x_{n-1} come from different threads, we have that $x_{n-1} \preceq_\pi x_0$ and hence, $time(f(x_{n-1})) \leq time(f(x_0))$. Therefore, we have that $time(f(x_0)) = time(f(x_{n-1}))$. This is not possible as x_0 and x_{n-1} correspond to different threads. Hence, every trace $\pi \in \mu$ is eliminated by $\bigwedge_{i=0}^{n-2} cons((x_i, x_{i+1}))$. □

Further, the minimal elimination cycle corresponds to a minimal constraint. As $A > R|X|^2$, atomic sections are used iff μ cannot be eliminated by reordering.

Theorem 3. *If $(\alpha, \sqsubseteq)$ is the constraint corresponding to the minimal cycle in the elimination graph, no strictly weaker constraint is sufficient to eliminate μ.*

Finding minimal cycles can be done by running an all-pairs shortest path algorithm, and finding nodes u, v from different threads such that sum of distances u to v and v to u is minimal. Hence, the theorem follows.

Theorem 4. *Finding minimal elimination cycles in the elimination graph $G(\mu, (\alpha, \sqsubseteq))$ can be done in time polynomial in the size of μ, α, and $\sqsubseteq$.*

7 Application to Systems Code

7.1 A Study of Concurrency Bugs in Linux Drivers

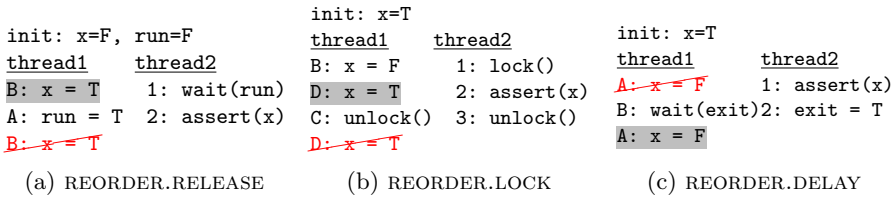
Our work is motivated by a study of concurrency defects in Linux device drivers. Drivers are required to perform well under concurrent workloads, which calls for sparing and fine-grained use of locks. This, in turn, provokes many concurrency-related bugs, making concurrency a major source of errors in drivers [4,12]. Our study considered 100 most recent (as of Dec. 2012) concurrency-related defects fixed in Linux device drivers (we used the Linux kernel development archive obtained from `www.kernel.org`). These defects occurred in 68 different drivers, all maintained by different developers. For each bug, we rely on manual code inspection to understand the exact nature of the bug and the fix.

We observed that many bug fixes involve subtle and seemingly ad hoc code transformations. In-depth analysis reveals several common patterns, shown in

Table 1. Synchronization patterns in Linux device drivers

pattern	description	#
REORDER	Reorder program statements to eliminate a race	28
LOCK	Protect racing code sections with a lock	17
OPTIMISTIC	Check if another thread has modified the value of a shared variable	10
BARRIER	Use a system-provided function to wait for a racing thread to terminate or complete a critical section	7
ATOMIC	Replace a statement-sequence with an equivalent atomic primitive	6
UPGRADE	Replace a synchronization primitive with a stronger one	5
UNSHARE	Avoid sharing by creating a private copy of a shared variable	3
CLONE	Replicate an idempotent statement	1
ADHOC	Transformations that do not fall into one of the previous categories	23
Total		**100**

Table 1. In particular, 28 of 100 fixes were semantic-preserving statement reorderings (the REORDER pattern). These further fall into several subpatterns (see Table 2 and Figure 3). Reordering instructions often involves additional side effects. For example, moving a statement across function boundaries may require adding arguments or return values to functions. Our implementation currently does not perform these, but can be extended to do so.



(a) REORDER.RELEASE (b) REORDER.LOCK (c) REORDER.DELAY

Fig. 3. Examples of REORDER subpatterns and corresponding elimination graphs

Interestingly, LOCK pattern (17%) is rarer than expected. Performance and kernel-imposed constraints often prevent lock usage. This observation confirms that locks are not a universal band-aid for concurrency defects in OS code. We do not discuss remaining bug categories, but note that we encountered 23 bug fixes that did not fit into any pattern (AD HOC in Table 1). We expect to discover new patterns among these as we include more defects in our study.

7.2 Synthesis Case Study

We implemented our algorithms in a tool called ConcurrencySwapper (Source and benchmarks can be found here: `https://github.com/thorstent/ConcurrencySwapper`). It handles a restricted subset of C, avoiding complex parts including pointer arithmetic, aliasing, bit-wise arithmetic, etc. It uses CPAChecker [2] to convert C statements into formulae representing instructions, as in Section 3. We use the bounded model checking tool Poirot [1] to detect three kinds of bugs: (a) assertion failures; (b) generic correctness

Table 2. Subpatterns of the REORDER pattern

pattern	description	example	#
REORDER.RELEASE	Move a variable assignment to a location before another thread accessing this variable is released	Fig 3a	11
REORDER.LOCK	Move statements to existing lock-protected section	Fig 3b	10
REORDER.DELAY	Delay assignment to a shared variable until a racing thread accessing this variable has terminated	Fig 3c	6
REORDER.RW	Reorder accesses to a pair of shared variables	Fig 1b	1
REORDER.ADHOC	Application-specific reordering	–	1

conditions (e.g., initialization-before-use for pointers); and (c) deadlocks (as Poirot does not detect deadlocks, we manually encoded these as suitable assertions for our examples). We generalize buggy traces, using Z3 theorem prover [5] to perform the required checks for Rules 1 and 2. The current implementation does not compute invariants during generalization; but even without invariant computation, our tool came up with the right program transformations quickly. To evaluate the effectiveness of trace generalization, we ran the experiments with and without it.

Reporting. Although each iteration of the algorithm eliminates a buggy po-trace, additional traces may exhibit the same bug. We report the iterations needed to completely fix a bug, i.e., until no more traces exhibit a similar bug. Also, we report separately, the time taken to: (a) find bugs; (b) generalize the trace and find a fix; and (c) verify the correct program. We report the verification time separately as it is usually the largest fraction of execution time.

Benchmarks. Our initial evaluation consisted of 5 microbenchmarks each of 15–30 lines of code without comments, and modeling a single concurrency defect found in a real Linux driver. The iterations required and fix patterns are summarized in Table 3. The synthesis took less than 15 seconds for each case, with trace analysis taking less than 0.5 seconds. Also, in 1 case, not using trace generalization leads to an additional iteration, leading to a larger execution time.

Table 3. Micro-benchmarks

Benchmark	Fix pattern	Iters.	Iters (w/o trace gen.)
ex1	REORDER.RW	1	1
ex2	REORDER.RELEASE	1	1
ex3	REORDER.LOCK	1	1
ex4	REORDER.ADHOC	3	3
ex5	LOCK	2	3

We evaluate the scalability of ConcurrencySwapper using a simplified version of the Linux Realtek 8169 driver. This driver is representative of medium to high-end drivers both in terms of overall complexity and the complexity of synchronization logic. We extracted the driver's complete synchronization skeleton, including code and variables related to thread synchronization and communication. The skeleton does not include the actual device management code, which is irrelevant to concurrency, and was additionally simplified to avoid currently unsupported C constructs. We provide an environment model to simulate all (7) OS threads that interact with the driver. The

Table 4. Results for Linux Realtek 8169 driver benchmark

Bug	Fix pattern	With trace generalization		Without trace generalization	
		Iters.	Bug-finding	Iters.	Additional bug-finding
bug1	REORDER.RELEASE	1	8 sec	1	same
bug2	REORDER.DELAY	1	23 sec	4	same + 80 sec
bug3	REORDER.RW	1	93 sec	1	same
bug4	REORDER.RW	1	94 sec	1	same
bug5	REORDER.ADHOC	2	47 sec	2	same

resulting skeleton had 364 LOC, while the original driver had around 7,000 LOC. The skeleton had 5 concurrency defects.

Poirot was able to find all the defects, and ConcurrencySwapper was able to find fixes for each defect through statement reordering. The results are summarized in Table 4. In each iteration, the trace analysis phase took less than 2 seconds. The extra bug finding times due to additional iterations is reported for the runs without trace generalization. In one case, the 3 additional iterations were required without trace generalization. The bug finding times dominate the trace analysis times, justifying the use of complex trace generalization procedure to avoid additional iterations. The verification phase took around 30 minutes.

8 Conclusion

The contributions of the paper are two-fold. First, our synthesis procedure considers a variety of semantics-preserving transformations (not just lock placement). Second, in order to speed up the synthesis procedure, we consider counterexamples that are partial orders on instructions (as opposed to linear orders). There are several possible directions for future work. First, we will investigate generalizations of counterexamples in CEGAR algorithms for verification, rather than synthesis. Second, we plan to address issues in systems programming such as weak memory models. In this paper, we compute minimal sufficient ordering constraints. In a sequentially-consistent models, reordering statements alone is sufficient to enforce the constraints; but in a weak model, additional fences may be needed to enforce them.

References

1. Poirot: The concurrency sleuth, `http://research.microsoft.com/en-us/projects/poirot/`
2. Beyer, D., Keremoglu, M.E.: CPAchecker: A tool for configurable software verification. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 184–190. Springer, Heidelberg (2011)
3. Cherem, S., Chilimbi, T., Gulwani, S.: Inferring locks for atomic sections. In: PLDI, pp. 304–315 (2008)
4. Chou, A., Yang, J., Chelf, B., Hallem, S., Engler, D.: An empirical study of operating systems errors. In: SOSP, pp. 73–88 (2001)

5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Ermis, E., Schäf, M., Wies, T.: Error invariants. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 187–201. Springer, Heidelberg (2012)
7. Gupta, A., Popeea, C., Rybalchenko, A.: Solving recursion-free horn clauses over LI+UIF. In: Yang, H. (ed.) APLAS 2011. LNCS, vol. 7078, pp. 188–203. Springer, Heidelberg (2011)
8. Gupta, A., Popeea, C., Rybalchenko, A.: Threader: A constraint-based verifier for multi-threaded programs. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 412–417. Springer, Heidelberg (2011)
9. Lipton, R.: Reduction: A method of proving properties of parallel programs. Commun. ACM 18(12), 717–721 (1975)
10. Mazurkiewicz, A.: Trace theory. In: Brauer, W., Reisig, W., Rozenberg, G. (eds.) APN 1986. LNCS, vol. 255, pp. 278–324. Springer, Heidelberg (1987)
11. Nielson, F., Nielson, H., Hankin, C.: Principles of program analysis (2. corr. print). Springer (2005)
12. Ryzhyk, L., Chubb, P., Kuz, I., Heiser, G.: Dingo: Taming device drivers. In: Eurosys (2009)
13. Solar-Lezama, A., Jones, C., Bodík, R.: Sketching concurrent data structures. In: PLDI, pp. 136–148 (2008)
14. Vechev, M., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI, pp. 125–135 (2008)
15. Vechev, M., Yahav, E., Yorsh, G.: Abstraction-guided synthesis of synchronization. In: POPL, pp. 327–338 (2010)

Multi-core Emptiness Checking of Timed Büchi Automata Using Inclusion Abstraction*

Alfons Laarman[1], Mads Chr. Olesen[2], Andreas Engelbredt Dalsgaard[2], Kim Guldstrand Larsen[2], and Jaco van de Pol[1]

[1] Formal Methods and Tools, University of Twente
{A.W.Laarman,J.C.vandePol}@utwente.nl
[2] Department of Computer Science, Aalborg University
{andrease,kgl,mchro}@cs.aau.dk

Abstract. This paper contributes to the multi-core model checking of timed automata (TA) with respect to liveness properties, by investigating checking of TA Büchi emptiness under the very coarse inclusion abstraction or zone subsumption, an open problem in this field.

We show that in general Büchi emptiness is not preserved under this abstraction, but some other structural properties are preserved. Based on those, we propose a variation of the classical nested depth-first search (NDFS) algorithm that exploits subsumption. In addition, we extend the multi-core CNDFS algorithm with subsumption, providing the first parallel LTL model checking algorithm for timed automata.

The algorithms are implemented in LTSMIN, and experimental evaluations show the effectiveness and scalability of both contributions: subsumption halves the number of states in the real-world FDDI case study, and the multi-core algorithm yields speedups of up to 40 using 48 cores.

1 Introduction

Model checking safety properties can be done with reachability, but only guarantees that the system does not enter a dangerous state, not that the system actually serves some useful purpose. To model check such liveness properties is more involved since they state conditions over infinite executions, e.g. that a request must infinitely often produce a result. One of the most well-known logics for describing liveness properties is Linear Temporal Logic (LTL) [2].

The automata-theoretic approach for LTL model checking [27] solves the problem efficiently by translating it to the Büchi emptiness problem, which has been shown decidable for real-time systems as well [1]. However, its complexity is exponential, both in the size of the system specification and of the property. In the current paper, therefore, we consider two possible ways of alleviating this so-called state space explosion problem: (1) by utilising the many cores in modern processors, and (2) by employing coarser abstractions to the state space.

* Danish authors partially supported by the MBAT ARTEMIS project, the MT-LAB VKR Centre of Excellence and the IDEA4CPS Sino-Danish Basic Research Centre.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 968–983, 2013.

Related work. The verification of timed automata was made possible by Alur and Dill's *region construction* [1], which represents clock valuations using constraints, called *regions*. A max-clock constant abstraction, or k-extrapolation, bounded the number of regions. Since the region construction is exponential in the number of clocks and constraints in the TA, coarser abstractions such as the symbolic *zone abstraction* have been studied [13], and also implemented in, among others, the state-of-the-art model checker UPPAAL [22]. Later, the k-extrapolation for zones was refined to include lower clock constraints in the so-called lower/upper-bound (LU) abstraction proposed in [4]. Finally, the *inclusion abstraction*, or simply *subsumption*, prunes reachability according to the partial order of the symbolic states [12]. All these abstractions preserve reachability properties [12,4].

Model checking LTL properties on timed automata, or equivalently checking timed Büchi automata (TBA) emptiness, was proven decidable in [1], by using the region construction. Bouajjani et al. [8] showed that the region-closed simulation graph preserve TBA emptiness. Tripakis [25] proved that the k-extrapolated zone simulation graph also preserves TBA emptiness, while posing the question whether other abstractions such as the LU abstraction and subsumption also preserve this property. Li [23] showed that the LU abstraction does in fact preserve TBA emptiness. The status of subsumption in LTL model checking is still open.

One way of establishing TBA emptiness on a finite simulation graph is the nested depth-first (NDFS) algorithm [9,16]. Recently, some multi-core version of these algorithms were introduced by Evangelista and Laarman et al [17,15,14]. These algorithms have the following properties: their runtime is linear in the number of states in the worst case while typically yielding good scalability; they are on-the-fly [18] and yield short counter examples [14, Sec. 4.3]. The latest version, called CNDFS, combines all these qualities and decreases memory usage [14].

In previous work, we parallelised reachability for timed automata using the mentioned abstractions [11]. It resulted in almost linear scalability, and speedups of up to 60 on a 48 core machine, compared to UPPAAL. The current work extends this previous work to the setting of liveness properties for timed automata. It also shares the UPPAAL input format, and re-uses the UPPAAL DBM library.

Problem statement. Parallel model checking of liveness properties for timed systems has been a challenge for several years. While advances were made with distributed versions of e.g. UPPAAL [3], these were limited to safety properties. Furthermore, it is unknown how subsumption, the coarsest abstraction, can be used for checking TBA emptiness.

Contributions. (1) For the first time, we realize parallel LTL model checking of timed systems using the CNDFS algorithm. (2) We prove that subsumption preserves several structural state space properties (Sec. 3), and show how these properties can be exploited by NDFS and CNDFS (Sec. 4 and Sec. 5). (3) We implement NDFS and CNDFS with subsumption in the LTSMIN toolset [20] and opaal [10]. Finally, (4) our experiments show considerable state space reductions by subsumption and good parallel scalability of CNDFS with speedups of up to 40 using 48 cores.

2 Preliminaries: Timed Büchi Automata and Abstractions

In the current section, we first recall the formalism of timed Büchi automata (TBA), that allows modelling of both a real-time system and its liveness requirements. Subsequently, we introduce finite symbolic semantics using zone abstraction with extrapolation and subsumption. Finally, we show which properties are known to be preserved under said abstractions.

Timed Automata and Transition Systems. Def. 2 provides a basic definition of a TBA. It can be extended with features such as finitely valued variables, and parallel composition to model networks of timed automata, as done in UPPAAL [5].

Definition 1 (Guards). *Let $\mathcal{G}(\mathcal{C})$ be a conjunction of clock constraints over the set of clocks $c \in \mathcal{C}$, generalized by:*

$$g ::= c \bowtie n \mid g \wedge g \mid true$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, =, >, \geq\}$ is a comparison operator. We call a guard downwards closed if all $\bowtie \in \{<, \leq, =\}$.

Definition 2 (Timed Büchi Automaton). *A timed Büchi automaton (TBA) is a 6-tuple $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, l_0, \rightarrow, I_{\mathcal{C}})$, where*

- *L is a finite set of locations, typically denoted by ℓ, where $\ell_0 \in L$ is the initial location, and $\mathcal{F} \subseteq L$, is the set of accepting locations,*
- *$\mathcal{C}$ is a finite set of clocks, typically denoted by c,*
- *$\rightarrow \subseteq L \times \mathcal{G}(\mathcal{C}) \times 2^{\mathcal{C}} \times L$ is the (non-deterministic) transition relation. We write $\ell \xrightarrow{g,R} \ell'$ for a transition, where ℓ is the source and ℓ' the target location, $g \in \mathcal{G}(\mathcal{C})$ is a transition guard, $R \subseteq \mathcal{C}$ is the set of clocks to reset, and*
- *$I_{\mathcal{C}} \colon L \rightarrow \mathcal{G}(\mathcal{C})$ is an invariant function, mapping locations to a set of guards. To simplify the semantics, we require invariants to be downwards-closed.*

The states of a TBA involve the notion of clock valuations. A clock valuation is a function $v : \mathcal{C} \rightarrow \mathbb{R}_{\geq 0}$. We denote all clock valuations over $\mathcal{C}$ with $\mathcal{V}_{\mathcal{C}}$. We need two operations on clock valuations: $v' = v + \delta$ for a delay of $\delta \in \mathbb{R}_{\geq 0}$ time units s.t. $\forall c \in \mathcal{C} \colon v'(c) = v(c) + \delta$, and reset $v' = v[R]$ of a set of clocks $R \subseteq \mathcal{C}$ s.t. $v'(c) = 0$ if $c \in R$, and $v'(c) = v(c)$ otherwise. We write $v \models g$ to mean that the clock valuation v satisfies the clock constraint g.

Definition 3 (Transition system semantics of a TBA). *The semantics of a TBA $\mathbb{B}$ is defined over the transition system $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ s.t.:*

1. *A state $s \in \mathcal{S}_v$ is a pair: (ℓ, v) with a location $\ell \in L$, and a clock valuation v.*
2. *An initial state $s_0 \in \mathcal{S}_v$, s.t. $s_0 = (\ell_0, v_0)$, where $\forall c \in \mathcal{C} \colon v_0(c) = 0$.*
3. *$\mathcal{T}_v \colon \mathcal{S}_v \times (\{\epsilon\} \cup \mathbb{R}_{\geq 0}) \times \mathcal{S}_v$ is a transition relation with $(s, a, s') \in \mathcal{T}_v$, denoted $s \xrightarrow{a} s'$ s.t. there are two types of transitions:*
   (a) *A discrete (instantaneous) transition: $(\ell, v) \xrightarrow{\epsilon} (\ell', v')$ if an edge $\ell \xrightarrow{g,R} \ell'$ exists, $v \models g$ and $v' = v[R]$, and $v' \models I_{\mathcal{C}}(\ell')$.*
   (b) *A delay by δ time units: $(\ell, v) \xrightarrow{\delta} (\ell, v + \delta)$ for $\delta \in \mathbb{R}_{\geq 0}$ if $v + \delta \models I_{\mathcal{C}}(\ell)$.*

We say a state $s \in \mathcal{S}$ is accepting, or $s \in \mathcal{F}$, when $s = (\ell, \ldots)$ and $\ell \in \mathcal{F}$. We write $s \xrightarrow{\delta}\xrightarrow{\epsilon} s'$ if there exists a state s'' such that $s \xrightarrow{\delta} s''$ and $s'' \xrightarrow{\epsilon} s'$. We denote an infinite run in $\mathcal{TS}_v^{\mathbb{B}} = (\mathcal{S}_v, s_0, \mathcal{T}_v)$ as an infinite path $\pi = s_1 \xrightarrow{\delta_1}\xrightarrow{\epsilon} s_2 \xrightarrow{\delta_2}\xrightarrow{\epsilon} s_3 \ldots$ The run is accepting if there exist an infinite number of indices i s.t. $s_i \in \mathcal{F}$. A(n infinite) run's time lapse is $Time(\pi) = \sum_{i \geq 1} \delta_i$. An infinite path π in $\mathcal{TS}_v^{\mathbb{B}}$ is *time convergent*, or zeno, if $Time(\pi) < \infty$, otherwise it is divergent. For example, the TBA in Fig. 1 has an infinite run: $(\ell_0, v_0) \xrightarrow{1} (\ell_0, v_0) \xrightarrow{1} \cdots$, that is not accepting, but is non-zeno. We claim that there is no accepting non-zeno run, exemplified by the finite run: $(\ell_0, v_0) \xrightarrow{2}\xrightarrow{\epsilon} (\ell_1, v_1) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_2, v_0) \xrightarrow{0}\xrightarrow{\epsilon} (\ell_1, v_0) \xrightarrow{1.9}\not\xrightarrow{\epsilon}$.

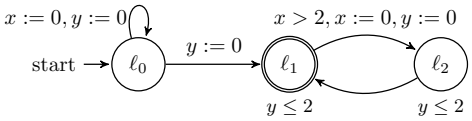


Fig. 1. A timed Büchi automaton

Definition 4 (A TBA's language and the emptiness problem). *The language accepted by $\mathbb{B}$, denoted $\mathcal{L}(\mathbb{B})$, is defined as the set of non-zeno accepting runs. The language emptiness problem for $\mathbb{B}$ is to check whether $\mathcal{L}(\mathbb{B}) = \emptyset$.*

Remark 1 (Zenoness). Zenoness is considered a modelling artifact as the behaviour it models cannot occur in any real system, which after all has finite processing speeds. Therefore, zeno runs should be excluded from analysis. However, any TBA $\mathbb{B}$ can be syntactically transformed to a *strongly non-zeno* $\mathbb{B}$' [26], s.t. $\mathcal{L}(\mathbb{B}) = \emptyset$ iff $\mathcal{L}(\mathbb{B}') = \emptyset$. Therefore, in the following, w.l.o.g., we assume that all TBAs are strongly non-zeno.

Definition 5 (Time-abstracting simulation relation). *A time-abstracting simulation relation R is a binary relation on $\mathcal{S}_v$ s.t. if $s_1 R s_2$ then:*
- *If $s_1 \xrightarrow{\epsilon} s_1'$, then there exists s_2' s.t. $s_2 \xrightarrow{\epsilon} s_2'$ and $s_1' R s_2'$.*
- *If $s_1 \xrightarrow{\delta} s_1'$, then there exists s_2' and δ' s.t. $s_2 \xrightarrow{\delta'} s_2'$ and $s_1' R s_2'$.*

If both R and R^{-1} are time-abstracting simulation relations, we call R a time-abstracting bisimulation relation.

Symbolic Abstractions using Zones. A *zone* is a symbolic representation of an infinite set of clock valuations by means of a clock constraint. These constraints are conjuncts (Def. 6) of simple linear inequalities on clock values, and thus describe (unbounded) convex polytopes in a $|\mathcal{C}|$-dimensional plane (e.g. Fig. 2). Therefore, zones can be efficiently represented by Difference Bounded Matrices (DBMs) [6].

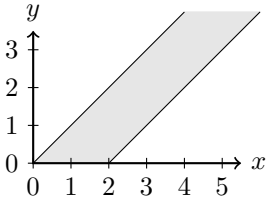


Fig. 2. A graphical representation of a zone over 2 clocks: $0 \leq x - y \leq 2$.

Definition 6 (Zones). *Similar to the guard definition, let $\mathcal{Z}(\mathcal{C})$ be the set of clock constraints over the set of clocks $c, c_1, c_2 \in \mathcal{C}$ generalized by:*

$$Z ::= c \bowtie n \mid c_1 - c_2 \bowtie n \mid Z \wedge Z \mid true \mid false$$

where $n \in \mathbb{N}_0$ is a constant, and $\bowtie \in \{<, \leq, >, \geq\}$ is a comparison operator. We also use $=$ for equalities, short for the conjunction of $\leq$ and $\geq$.

We write $v \models Z$ if the clock valuation v is included in Z, for the set of clock valuations in a zone $[\![Z]\!] = \{v \mid v \models Z\}$, and for *zone inclusion* $Z \subseteq Z'$ iff $[\![Z]\!] \subseteq [\![Z']\!]$. Notice that $[\![false]\!] = \emptyset$. Using the fundamental operations below, which are detailed in [6], we define the *zone semantics* over *simulation graphs* in Def. 7. Most importantly, these operations are implementable in $O(n^3)$ or $O(n^2)$ and closed w.r.t. $\mathcal{Z}$.
delay: $[\![Z \uparrow]\!] = \{v + \delta | \delta \in \mathbb{R}_{\geq 0}, v \in [\![Z]\!]\}$,
clock reset: $[\![Z[R]]\!] = \{v[R] | v \in [\![Z]\!]\}$, and
constraining: $[\![Z \wedge Z']\!] = [\![Z]\!] \cap [\![Z']\!]$.

Definition 7 (Zone semantics). *The semantics of a TBA* $\mathbb{B} = (L, \mathcal{C}, \mathcal{F}, \ell_0, \rightarrow, I_{\mathcal{C}})$ *under the zone abstraction is a simulation graph:* $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ *s.t.:*

1. $\mathcal{S}_{\mathcal{Z}}$ *consists of pairs* (ℓ, Z) *where* $\ell \in L$*, and* $Z \in \mathcal{Z}$ *is a zone.*
2. $s_0 \in \mathcal{S}_{\mathcal{Z}}$ *is an initial state* $(\ell_0, Z_0 \uparrow \wedge\ I_{\mathcal{C}}(\ell_0))$ *with* $Z_0 = \bigwedge_{c \in \mathcal{C}} c = 0$.
3. $\mathcal{T}_{\mathcal{Z}}$ *is the symbolic transition function using zones, s.t.* $(s, s') \in \mathcal{T}_{\mathcal{Z}}$*, denoted* $s \Rightarrow s'$ *with* $s = (\ell, Z)$ *and* $s' = (\ell', Z')$*, if an edge* $\ell \xrightarrow{g,R} \ell'$ *exists, and* $Z \wedge g \neq false$*,* $Z' = (((Z \wedge g)[R]) \uparrow) \wedge\ I_{\mathcal{C}}(\ell')$ *and* $Z' \neq false$*.*

Any simulation graph is a discrete graph, hence cycles and lassos are defined in the standard way. We write $s \Rightarrow^+ s'$ iff there is a non-empty path in $SG(\mathbb{B})$ from s to s', or $s \Rightarrow^* s'$ if the path can be empty. An infinite run in $SG(\mathbb{B})$ is an infinite sequence of states $\pi = s_1 s_2 \ldots$, s.t. $s_i \Rightarrow s_{i+1}$ for all $i \geq 1$. It is accepting if it contains infinitely many accepting states. If $SG(\mathbb{B})$ is finite, any infinite path from s_0 defines a lasso: $s_0 \Rightarrow^* s \Rightarrow^+ s$.

Definition 8 (A TBA's language under Zone Semantics). *The language accepted by a TBA* $\mathbb{B}$ *under the zone semantics, denoted* $\mathcal{L}(SG(\mathbb{B}))$*, is the set of infinite runs* $\pi = s_0 s_1 s_2 \ldots$ *s.t. there exists an infinite set of indices s.t.* $s_i \in \mathcal{F}$*.*

Because there are infinitely many zones, the state space of $SG(\mathbb{B})$ may also be infinite. To bound the number of zones, *extrapolation* methods combine all zones which a given TBA $\mathbb{B}$ cannot distinguish. For example, k-extrapolation finds the largest upper bound k in the guards and invariants of $\mathbb{B}$, and extrapolates all bounds in the zones $\mathcal{Z}$ that exceed this value, while LU-extrapolation uses both the maximal lower bound l and the maximal upper bound u [4]. Extrapolation can be refined on a per-clock basis [4], and on a per-location basis.

Definition 9. *An abstraction over a simulation graph* $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ *is a mapping* $\alpha : \mathcal{S}_{\mathcal{Z}} \rightarrow \mathcal{S}_{\mathcal{Z}}$ *s.t. if* $\alpha((\ell, Z)) = (\ell', Z')$ *then* $\ell = \ell'$ *and* $Z \subseteq Z'$*.*
If the image of an abstraction α *is finite, we call it a finite abstraction.*

Definition 10. *Abstraction* α *over zone transition system* $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ *induces a zone transition system* $SG_\alpha(\mathbb{B}) = (\mathcal{S}_\alpha, \alpha(s_0), \mathcal{T}_\alpha)$ *where:*

- $\mathcal{S}_\alpha = \{\alpha(s) \mid s \in \mathcal{S}_{\mathcal{Z}}\}$ *is the set of states, s.t.* $\mathcal{S}_\alpha \subseteq \mathcal{S}_{\mathcal{Z}}$*,*
- $\alpha(s_0)$ *is the initial state, and*
- $(s, s') \in \mathcal{T}_\alpha$ *iff* $(s, s'') \in \mathcal{T}_{\mathcal{Z}}$ *and* $s' = \alpha(s'')$*, is the transition relation.*

We call an abstraction α an *extrapolation* if there exists a simulation relation R s.t. if $\alpha((\ell, Z)) = (\ell, Z')$ then for all $v' \in Z'$ there exist a $v \in Z$ s.t. $v' R v$ [23]. This means extrapolations do not introduce behaviour that the un-extrapolated system cannot simulate. The abstraction defined by k-extrapolation is denoted by α_k, while the abstraction defined by LU-extrapolation is called α_{lu}. Hence, α_k and α_{lu} induce finite simulation graphs, written $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$.

Subsumption abstraction. While $SG_k(\mathbb{B})$ and $SG_{lu}(\mathbb{B})$ are finite, their size is still exponential in the number of clocks. Therefore, we turn to the coarser inclusion/ subsumption abstraction of [12], hereafter denoted *subsumption abstraction*. We extend the notion of subsumption to states: a state $s = (\ell, Z) \in \mathcal{S}_{\mathcal{Z}}$ is *subsumed* by another $s' = (\ell', Z')$, denoted $s \sqsubseteq s'$, when $\ell = \ell'$ and $Z \subseteq Z'$. Let $\mathcal{R}(SG(\mathbb{B})) = \{s | s_0 \Rightarrow^* s\}$ denote the set of *reachable states* in $SG(\mathbb{B})$.

Proposition 1 ($\sqsubseteq$ is a simulation relation). *If $(\ell, Z_1) \sqsubseteq (\ell, Z_2)$ and $(\ell, Z_1) \Rightarrow (\ell', Z_1')$ then there exists Z_2' s.t. $(\ell, Z_2) \Rightarrow (\ell', Z_2')$ and $(\ell', Z_1') \sqsubseteq (\ell', Z_2')$.*

Proof. By the definition of $\sqsubseteq$, and the fact that $\Rightarrow$ is monotone w.r.t $\subseteq$ of zones.

Definition 11 (Subsumption abstraction [12]). *A subsumption abstraction $\alpha_\sqsubseteq$ over a zone transition system $SG(\mathbb{B}) = (\mathcal{S}_{\mathcal{Z}}, s_0, \mathcal{T}_{\mathcal{Z}})$ is a total function $\alpha_\sqsubseteq : \mathcal{R}(SG(\mathbb{B})) \to \mathcal{R}(SG(\mathbb{B}))$ s.t. $s \sqsubseteq \alpha_\sqsubseteq(s)$*

Note the subsumption abstraction is defined only over the reachable state space, and is *not* an extrapolation, because it might introduce extra transitions that the unabstracted system cannot simulate. Typically α is constructed on-the-fly during analysis, only abstracting to states that are already found to be reachable. This makes its performance depend heavily on the search order, as finding "large" states quickly can make the abstraction coarser [11].

Property preservation under abstractions. We now consider the preservation by the abstractions above of the property of *location reachability* (a location ℓ is reachable iff $s_0 \Rightarrow^* (\ell, \ldots)$) and that of Büchi emptiness.

Proposition 2. *For any of the abstractions α: α_k [12], α_{lu} [4], $\alpha_k \circ \alpha_\sqsubseteq$ [12], and $\alpha_{lu} \circ \alpha_\sqsubseteq$ [4], it holds that ℓ is reachable in $\mathcal{TS}_v^{\mathbb{B}} \iff \ell$ is reachable in $SG_\alpha(\mathbb{B})$*

Proposition 3. *For any finite extrapolation [23] α, e.g. the abstractions α_k [25] and α_{lu} [23] it holds that $\mathcal{L}(\mathbb{B}) = \emptyset \iff \mathcal{L}(SG_\alpha(\mathbb{B})) = \emptyset$*

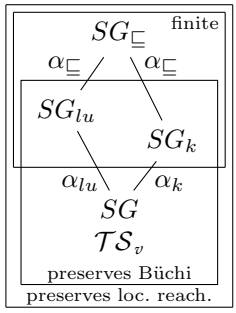


Fig. 3. Abstractions

From hereon we will denote any finite extrapolation as α_{fin}, and the associated simulation graph $SG_{fin}(\mathbb{B})$. To denote that this graph can be generated *on-the-fly* [27,2,12], we use a NEXT-STATE(s) function which returns the set of successor states for s: $\{s' \in \mathcal{S}_{fin} \mid s \Rightarrow s'\}$.

As a result of Prop. 3 we can focus on finding accepting runs in $SG_{fin}(\mathbb{B})$. Because it is finite, any such run is represented by a lasso: $s_0 \Rightarrow s \Rightarrow^+ s$. Tripakis [25] poses the question of whether $\alpha_\sqsubseteq$ can be used to check Büchi emptiness. We will investigate this further in the next section.

3 Preservation of Büchi Emptiness under Subsumption

The current section, investigates what properties are preserved by a subsumption abstraction $\alpha_{\sqsubseteq}$, when applied on a finite simulation graph obtained by an extrapolation, α_{fin}, in the following, denoted as $SG_{\sqsubseteq}(\mathbb{B}) = (SG_{fin \circ \sqsubseteq}(\mathbb{B}))$.

Proposition 4. *For all abstractions α, $s \in \mathcal{F} \Leftrightarrow \alpha(s) \in \mathcal{F}$ (by Def. 9).*

Proposition 5. *An $\alpha_{\sqsubseteq}$ abstraction is safe w.r.t. Büchi emptiness:*

$$\mathcal{L}(\mathbb{B}) \neq \emptyset \implies \mathcal{L}(SG_{\sqsubseteq}(\mathbb{B})) \neq \emptyset$$

Proof. If $\mathcal{L}(\mathbb{B}) \neq \emptyset$, there must be an infinite accepting path π. This path is inscribed [25] in $SG_{fin}(\mathbb{B})$, and because $\sqsubseteq$ is a simulation relation a similar path exists in $SG_{\sqsubseteq}(\mathbb{B})$. □

Prop. 5 shows that subsumption abstraction preserves Büchi emptiness in one direction. Unfortunately, an accepting cycle in $SG_{\sqsubseteq}(\mathbb{B})$ is not always reflected in $SG_{fin}(\mathbb{B})$, as Fig. 4 illustrates. The figure visualises $SG_{\sqsubseteq}(\mathbb{B})$ by drawing subsumed states inside subsuming states (e.g. $s_3 \sqsubseteq s_1$).

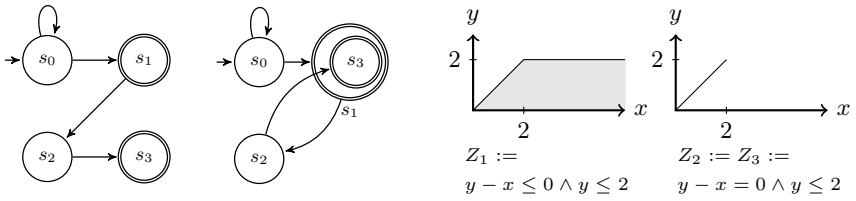


Fig. 4. The state space $SG_{\sqsubseteq}(\mathbb{B})$ of the model in Fig. 1 with $\ell_1 \in \mathcal{F}$ contains 4 states (shown on the left): s_0, $s_1 = (\ell_1, Z_1)$, $s_2 = (\ell_2, Z_2)$ and $s_3 = (\ell_1, Z_3)$. The graphical representation of the zones Z_1–Z_3 (right) reveals that $Z_3 \subseteq Z_1$ and hence $s_3 \sqsubseteq s_1$. As $s_3 \sqsubseteq s_1$ and both are reachable, a subsumption abstraction is allowed to map $\alpha_{\sqsubseteq}(s_3) = s_1$, introducing a cycle $s_1 \Rightarrow s_2 \Rightarrow s_1$ in $SG_{\sqsubseteq}(\mathbb{B})$.

However, subsumption introduces strong properties on paths and cycles to which we devote the rest of the current section. In subsequent sections, we exploit these properties to improve algorithms that implement the TBA emptiness check.

Lemma 1 (Accepting cycles under $\sqsubseteq$). *If $SG_{fin}(\mathbb{B})$ contains states s, s' s.t. s leads to an accepting cycle and $s \sqsubseteq s'$, then s' leads to an accepting cycle.*

Proof. We have that $s \Rightarrow^* t \Rightarrow^+ t$, and because $\sqsubseteq$ is a simulation relation we have the existence of a state x s.t. $t \sqsubseteq x$:

$$\begin{array}{ccccccc} s' & \Rightarrow^* & t' & \Rightarrow & \cdots & \Rightarrow & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| \\ s & \Rightarrow^* & t & \Rightarrow & \cdots & \Rightarrow & t \end{array}$$

From x, we again have a similar path, to some x'. This sequence will eventually repeat some x'', because $SG_{fin}(\mathbb{B})$ is finite. It follows that all states in $x'' \Rightarrow^+ x''$ subsume states in $t \Rightarrow^+ t$, hence the former cycle is also accepting (Prop. 4). □

Lemma 2 (Paths under $\sqsubseteq$). *If $SG_{fin}(\mathbb{B})$ contains a path $s \Rightarrow^+ s'$ containing an accepting state and $s \sqsubseteq s'$, then s leads to an accepting cycle.*

Proof. Because $\sqsubseteq$ is a simulation relation we have that $s \Rightarrow^+ s'$ and $s \sqsubseteq s'$ implies the existence of some t such that $s' \Rightarrow^+ t$ and $s' \sqsubseteq t$. From t, we again obtain a similar path to some t', s.t. $t \sqsubseteq t'$. Because $SG_{fin}(\mathbb{B})$ is finite, the sequence of t's will eventually repeat some element x, s.t. $x \Rightarrow^+ \cdots \Rightarrow^+ x$.

$$\begin{array}{ccccccccccc} s' & \Rightarrow^+ & t & \Rightarrow^+ & t' & \Rightarrow^+ & \cdots & \Rightarrow^+ & t'' & \Rightarrow^+ & x \\ \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \sqcup\!| & & \| \\ s & \Rightarrow^+ & s' & \Rightarrow^+ & t & \Rightarrow^+ & \cdots & \Rightarrow^+ & x & \Rightarrow^+ & x \end{array}$$

This gives us the lasso $s \Rightarrow^* x \Rightarrow^+ x$. It also follows that all states in $x \Rightarrow^+ x$ subsume states in $s \Rightarrow^+ s'$, hence the former cycle is accepting (Prop. 4). □

4 Timed Nested Depth-First Search with Subsumption

In the current section, we extend the classic linear-time NDFS [9,24] algorithm to exploit subsumption. The algorithm detects accepting cycles, the absence of which implies Büchi emptiness. It is correct for the graph $SG_{fin}(\mathbb{B})$ according to Prop. 3. In the following, with *soundness*, we mean that when NDFS reports a cycle, indeed an accepting cycle exists in the graph, while completeness indicates that NDFS always reports an accepting cycle if the graph contains one.

The NDFS algorithm in Alg. 1 consists of an outer DFS (*dfsBlue*) that sorts accepting states s in DFS *postorder*. And an inner DFS (*dfsRed*) that searches for cycles over each s, called the *seed*. States are maintained in 3 colour sets:

Alg. 1. NDFS

```
1: procedure ndfs( )
2:     Cyan := Blue := Red := ∅
3:     dfsBlue(s_0)
4:     report no cycle
5: procedure dfsRed(s)
6:     Red := Red ∪ {s}
7:     for all t in NEXT-STATE(s) do
8:         if (t ∈ Cyan) then report cycle
9:         if (t ∉ Red) then dfsRed(t)
10: procedure dfsBlue(s)
11:     Cyan := Cyan ∪ {s}
12:     for all t in NEXT-STATE(s) do
13:         if t ∉ Blue ∧ t ∉ Cyan then
14:             dfsBlue(t)
15:     if s ∈ F then
16:         dfsRed(s)
17:     Blue := Blue ∪ {s}
18:     Cyan := Cyan \ {s}
```

1. *Blue*, states *explored* by *dfsBlue*,
2. *Cyan*, states on the stack of *dfsBlue* (*visited* but not yet explored), which are used by *dfsRed* to close cycles over s early at l.8 [24], and
3. *Red*, visited by *dfsRed*.

Alg. 1 maintains a few strong invariants, which are already mentioned in [9,24]:

I0: At l.13 all red states are blue.
I1: The only accepting state visited by *dfsRed* is the seed.
I2: Outside of *dfsRed*, accepting cycles are not reachable from red states.
I3: A sufficient postcondition for *dfsRed*(s) is that all reachable states from s are included in *Red* and no cyan state is reachable from it.

We now try to employ subsumption on the different colours to prune the searches, even though we cannot use it on all colours as $SG_{\sqsubseteq}(\mathbb{B})$ introduces additional cycles as Fig. 4 showed. To express subsumption checks on sets we write $s \sqsubseteq S$, meaning $\exists s' \in S\colon s \sqsubseteq s'$. And $S \sqsubseteq s$, meaning $\exists s' \in S\colon s' \sqsubseteq s$. At several places in Alg. 1 we might apply subsumption, leading to the following options:

1. On cyan for cycle detection:
   (a) $t \sqsubseteq \mathit{Cyan}$ at l.8, or
   (b) $\mathit{Cyan} \sqsubseteq t$ at l.8.
2. On *dfsBlue*, by replacing $t \notin \mathit{Blue} \wedge t \notin \mathit{Cyan}$ at l.13 with $t \not\sqsubseteq \mathit{Blue} \cup \mathit{Cyan}$.
3. On the blue set (explored states), by replacing $t \notin \mathit{Blue}$ at l.13 with $t \not\sqsubseteq \mathit{Blue}$.
4. On *dfsRed*, by replacing $t \notin \mathit{Red}$ at l.9 with $t \not\sqsubseteq \mathit{Red}$.

Subsumption on cyan for cycle detection as in option 1a makes the algorithm unsound: cycles in $SG_{\sqsubseteq}(\mathbb{B})$ are not always reflected in $SG_{fin}(\mathbb{B})$ (Fig. 4). There is also no hope of "unwinding" the algorithm upon detecting an accepting cycle that does not exist in the underlying $SG_{fin}(\mathbb{B})$ without losing its linear-time complexity, as the number of cycles can be exponential in the size of $SG_{\sqsubseteq}(\mathbb{B})$.

If, on the other hand, we prune the blue search as in option 2, the algorithm becomes incomplete. Fig. 5 shows a run of the modified NDFS on an $SG_{fin}(\mathbb{B})$ with cycle $s_3 \Rightarrow s_2 \Rightarrow s_3$. The *dfsBlue* backtracked over s_2 as $s_3 \sqsubseteq s_1$ and $s_1 \in \mathit{Cyan}$. The *dfsRed* now launched from s_1, will however continue to visit s_3, while missing the cycle as s_2 is not cyan. We also observe that I1 is violated, indicating that the postorder on accepting states (s_3 before s_1) is lost.

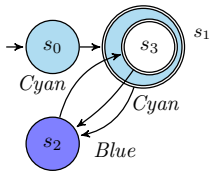
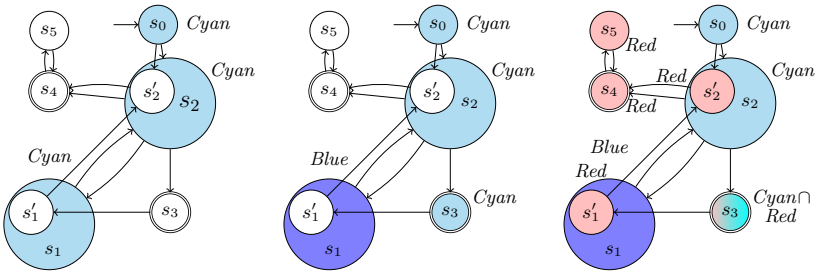


Fig. 5. Counter example to subsumption on *Blue* and *Cyan* (option 2).

It is tempting therefore to use subsumption on blue only, as in option 3. However, Fig. 6 shows an "animation" of a run with the modified NDFS which is incomplete. Here state s_1 is first backtracked in the blue search as all successors are cyan (left). Then state s_1 is marked blue; The blue search backtracks to s_2, proceeds to s_3 and backtracks because it finds $s_1' \sqsubseteq s_1 \in \mathit{Blue}$ (middle). Then a red search is started from s_3, which subsumes the cyan stack (s_2) and visits accepting state s_4, violating I1 and missing the accepting cycle $s_4 \Rightarrow s_5 \Rightarrow s_4$.



(a) *dfsBlue*(s_1) (b) *dfsBlue*(s_3) (c) *dfsRed* from s_3

Fig. 6. Counter example to subsumption on *Blue*

A viable option however is to use inverse subsumption on cyan as in option 1b. According to Lemma 1, a state that subsumes a state on the cyan stack leads to a cycle. And as the only goal of the red search is to find a cyan state (to close an accepting cycle over the seed), it does not rely on DFS (I3). Thus we may as well use subsumption in the red search as in option 4. By definition (Def. 11), $SG_{\sqsubseteq}(\mathbb{B})$ contains a "larger" state for all reachable states in $SG_{fin}(\mathbb{B})$. So in combination with option 1b this is sufficient to find all accepting cycles.

The strong invariant (I2) states accepting cycles are not reachable from red states, so red states can prune the blue search. We can strengthen the condition on l.13 to $t \notin \mathit{Blue} \cup \mathit{Cyan} \cup \mathit{Red}$. However, this is of no use since by (I0), $\mathit{Red} \subseteq \mathit{Blue}$. Luckily, even states subsumed by red do not lead to accepting cycles (contraposition of Lemma 1), so we can use subsumption again: $t \notin \mathit{Blue} \cup \mathit{Cyan} \wedge t \not\sqsubseteq \mathit{Red}$. The benefit of this can be illustrated using Fig. 4. Once *dfsBlue* backtracks over s_1, we have $s_1, s_2, s_3 \in \mathit{Red}$ by *dfsRed* at l.16. Any hypothetical other path from s_0 to a state subsumed by these red states can be ignored.

Alg. 2 shows a version of NDFS with all correct improvements. Notice that I2 and I3 are sufficient to conclude correctness of these modifications.

Alg. 2. NDFS with subsumption on red, cycle detection, and red prune of *dfsBlue*.

```
1: procedure ndfs( )
2:     Cyan := Blue := Red := ∅
3:     dfsBlue(s_0)
4:     report no cycle
5: procedure dfsRed(s)
6:     Red := Red ∪ {s}
7:     for all t in NEXT-STATE(s) do
8:         if (Cyan ⊑ t) then report cycle
9:         if (t ⋢ Red) then dfsRed(t)
10: procedure dfsBlue(s)
11:     Cyan := Cyan ∪ {s}
12:     for all t in NEXT-STATE(s) do
13:         if (t ∉ Blue ∪ Cyan ∧ t ⋢ Red)
14:         then dfsBlue(t)
15:     if s ∈ F then
16:         dfsRed(s)
17:     Blue := Blue ∪ {s}
18:     Cyan := Cyan \ {s}
```

5 Multi-core CNDFS with Subsumption

CNDFS [14] is a parallel algorithm for checking Büchi emptiness [14]. By Prop. 3, it is correct for SG_{fin}. In the current section, we extend CNDFS with subsumption, in a similar way as we have done for the sequential NDFS in the previous section.

In CNDFS (Alg. 3 without underlined parts), each worker thread i runs a seemingly independent $dfsBlue_i$ and $dfsRed_i$, with a local stack colour $Cyan_i$, and its own random successor ordering (indicated by the subscript i of the NEXT-STATE function). However, the workers assist each other by sharing the colours *Blue* and *Red* globally, thus pruning each other's search space.

The main problem that CNDFS has to solve is the loss of postorder on the accepting states due to the shared blue color (similar to the effects of option 3 as illustrated in Fig. 6). In the previous section, we have seen that a loss of postorder may cause *dfsRed* to visit non-seed accepting states, i.e. I1 is violated. CNDFS demonstrates that repairing the latter *dangerous situation* is sufficient to preserve correctness [14, Sec. 3].

To detect a dangerous situation, CNDFS collects the states visited by $dfsRed_i$ in a set $\mathcal{R}_i$ at l.7. After completion of $dfsRed_i$, the algorithm then checks $\mathcal{R}_i$ for non-seed accepting states at l.21. By simply waiting for these states to become red, the dangerous situation is resolved as the blue state that caused the situation was always placed by some other worker, which will eventually continue [14, Prop. 3]. Once the situation is detected to be resolved, all states from the local $\mathcal{R}_i$ are added to *Red* at l.22.

CNDFS maintains similar invariants as NDFS:

I2' Red states do not lead to accepting cycles (Lemma 1 and Prop. 2 in [14]).
I3' After $dfsRed_i(s)$ states reachable from s are red or in $\mathcal{R}_i$ (from [14, Lem. 2]).

Because these invariants are as strong or stronger than I2 and I3, we can use subsumption in a similar way as for NDFS. Alg. 3 underlines the changes to algorithm w.r.t. Alg. 2 in [14]. We additionally have to extend the waiting procedure

Alg. 3. CNDFS with subsumption

```
1: procedure cndfs(P)
2:     Blue := Red := ∅
3:     forall i in 1..P do Cyan_i := ∅
4:     dfsBlue_1(s_0) ‖ .. ‖ dfsBlue_P(s_0)
5:     report no cycle
6: procedure dfsRed_i(s)
7:     R_i := R_i ∪ {s}
8:     for all t in NEXT-STATE_i(s) do
9:         if Cyan ⊑ t then cycle
10:        if t ∉ R_i ∧ t ⋢ Red then
11:            dfsRed_i(t)
12: procedure dfsBlue_i(s)
13:    Cyan_i := Cyan_i ∪ {s}
14:    for all t in NEXT-STATE_i(s) do
15:        if t ∉ Cyan_i ∪ Blue ∧ t ⋢ Red then
16:            dfsBlue(t)
17:    Blue := Blue ∪ {s}
18:    if s ∈ F then
19:        R_i := ∅
20:        dfsRed(s)
21:        await ∀s' ∈ R_i ∩ F \ {s}: s' ⊑ Red
22:        forall s' in R_i do Red := Red ∪ s'
23:    Cyan_i := Cyan_i \ {s}
```

to include subsumption at l.21, because the use of subsumption in $dfsRed_i$ can cause other workers to find "larger" states.

In the next section, we will benchmark Alg. 3 on timed models. An important property that the algorithm inherits from CNDFS, is that its *runtime* is linear in the size of the input graph N. However, in the worst case, all workers may visit the same states. Therefore, the complexity of the amount of *work* that the algorithm performs (or the amount of power it consumes) equals $N \times P$, where P is the number of processors used. The randomised successor function NEXT-STATE$_i$ however ensures that this does not happen for most practical inputs. Experiments on over 300 examples confirmed this [14, Sec. 4], making CNDFS the current state-of-the-art parallel LTL model checking algorithm.

6 Experimental Evaluation

To evaluate the performance of the proposed algorithms experimentally, we implemented CNDFS without [14] and with subsumption (Alg. 3) in LTSMIN 2.0[1]. The `opaal` [10] tool[2] functions as a front-end for UPPAAL models. Previously, we demonstrated scalable multi-core reachability for timed automata [11].

Experimental setup. We benchmarked[3] on a 48-core machine (a four-way AMD Opteron™ 6168) with a varying number of threads, averaging results over 5 repetitions. We consider the following models and LTL properties:

`csma`[4] is a protocol for Carrier Sense, Multiple-Access with Collision Detection with 10 nodes. We verify the property that on collisions, eventually the bus will be active again: $\Box$(`(P0=bus_collision1)` $\Longrightarrow$ $\Diamond$`(P0=bus_active)`).

`fischer-1/2`[5] implements a mutual exclusion protocol; a canonical benchmark for timed automata, with 10 nodes. As in [23], we use the property (1): $\neg$(`(`$\Box\Diamond$`k=1)`$\vee$`(`$\Box\Diamond$`k=0)`), where `k` is the number of processes in their critical section. We also add a weak fairness property (2): $\Box$(`(`$\Box$`P_1=req)` $\Longrightarrow$ `(`$\Diamond$`P_1=cs)`): processes requesting infinitely often will eventually be served.

`fddi`[4] models a token ring system as described in [8], where a network of 10 stations are organised in a ring and can hand back the token in a synchronous or asynchronous fashion. We verify the property from [8] that every station will eventually send asynchronous messages: $\Box$(`(`$\Diamond$`(ST1=station_z_sync))`.

`train-gate`[4] models a railway interlocking, with 10 trains. Trains drive onto the interconnect until detected by sensors. There they wait until receiving a signal for safe crossing. The property prescribes that each approaching train eventually should be serviced: $\Box$`(Train_1=Appr` $\Longrightarrow$ `(`$\Diamond$`Train_1=Cross))`.

The following command-line was used to start the LTSMIN tool: `opaal2lts-mc --strategy=[A] --ltl-semantics=textbook --ltl=[f] -s28 --threads=[P] -u[0,1] [m]`.

[1] Available as open source at: `http://fmt.cs.utwente.nl/tools/ltsmin`

[2] Available as open source at: `http://opaal-modelchecker.com`

[3] All results are available at: `http://fmt.cs.utwente.nl/tools/ltsmin/cav-2013`

[4] From `http://www.it.uu.se/research/group/darts/uppaal/benchmarks/`

[5] As distributed with UPPAAL.

This runs algorithm A on the cross-product of the model m with the Büchi automaton of formula f. It uses a fixed hash table of size 2^{28} and P threads, and either subsumption (-u1) or not (-u0). The option ltl-semantics selects textbook LTL semantics as defined in [2, Ch. 4]. To investigate the overhead of CNDFS, we also run the multi-core algorithms for plain reachability on this crossproduct, even though this does not make sense from a model checking perspective. To compare effects of the search order on subsumption, we use both DFS and BFS.

Note finally, that we are only interested here in full verification, i.e. in LTL properties that are correct w.r.t the system under verification. This is the hardest case as the algorithm has to explore the full simulation graph. To test their on-the-fly nature, we also tried a few incorrect LTL formula for the above models, to which the algorithms all delivered counter examples within a second. But with parallelism this happens almost instantly [14, Sec. 4.2].

Implementation. LTSMIN defines a NEXT-STATE function as part of its PINS interface for language-independent symbolic/parallel model checking [7]. Previously, we extended PINS with subsumption [11]. opaal is used to parse the UPPAAL models and generate C code that implements PINS. The generated code uses the UPPAAL DBM library to implement the simulation graph semantics under *LU-extrapolated zones*. The LTL crossproduct [2] is calculated by LTSMIN.

LTSMIN's multi-core tool [20] stores states in one lockless hash/tree table in shared memory [19,21]. For timed systems, this table is used to store *explicit state parts*, i.e. the locations and state variables [5]. The DBMs representing zones, here referred to as the *symbolic state parts*, are stored in a separate lockless hash table, while a lockless *multimap* structure efficiently stores full states, by linking multiple symbolic to a single explicit state part [11]. Global colour sets of CNDFS (*Blue* and *Red*) are encoded with extra bits in the multimap, while local colours are maintained in local tables to reduce contention to a minimum.

Hypothesis. CNDFS for untimed model checking scaled mostly linearly. In the timed automata setting, several parameters could change this picture. In the first place, the *computational intensity* increases, because the DBM operations use many calculations. In modern multi-core computers, this feature improves scalability, because it more closely matches the machine's high frequency/bandwidth ratio [19]. On the other hand, the lock granularity increases since a single lock now governs multiple DBMs stored in the multimap [11, Sec. 6.1]. Nonetheless, for multi-core timed reachability, previous experiments showed almost linear scalability [11, Sec. 7], even when using other model checkers (UPPAAL) as a base line. On the other hand, the CNDFS algorithm requires more queries on the multimap structure to distinguish the different colour sets.

Subsumption probably improves the absolute performance of CNDFS. We expect that models with many clocks and constraints exhibit a better reduction than others. Moreover, it is known [3] that the reduction due to subsumption depends strongly on the exploration order: BFS typically results in better reductions than DFS, since "large" states are encountered later. CNDFS might share this disadvantage with DFS. However, as shown in [11], subsumption with

Table 1. Runtimes (sec) and states counts *without* subsumption

Model	P	$\|L\|$	$\|\mathcal{R}\|$	$\|V\|_{cndfs}$	$\|\Rightarrow\|_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	135449	438005	438005	1016428	26.1	26.2	27.8
csma	48	135449	438005	453658	1016428	1.0	0.9	0.9
fddi	1	119	179515	179515	314684	26.3	26.6	34.2
fddi	48	119	179515	566093	314684	1.6	0.7	2.7
fischer-1	1	521996	4987796	4987796	19481530	195.9	196.7	212.2
fischer-1	48	521996	4987796	5190490	19481530	4.8	4.6	5.1
fischer-2	1	358901	3345866	3345866	10426444	135.8	136.5	145.5
fischer-2	48	358901	3345866	3541373	10426444	3.4	3.3	3.7
train-gate	1	119989268	119989268	119989268	177201017	1608	1621	1724
train-gate	48	119989268	119989268	319766765	177201017	34.9	45.4	145.8

random parallel DFS performs much better than sequential DFS, which could be beneficial for the scalability of CNDFS. So it is really hard to predict the relative performance and scalability of these algorithms, and the effects of subsumption.

Experimental results without subsumption. We first compare the algorithms BFS, DFS (parallel reachability) and CNDFS (accepting cycles) without subsumption. Table 1 shows their sequential ($P = 1$) and parallel ($P = 48$) runtimes (T). Note that sequential CNDFS is just NDFS. We show the number of explicit state parts ($|L|$), full states ($|\mathcal{R}|$), transitions ($|\Rightarrow|$), and also the number of states visited in CNDFS ($|V|$). These numbers confirm the findings reported previously for CNDFS applied to untimed systems: The sequential run times ($P = 1$) are very similar, indicating little overhead in CNDFS. For the parallel runs ($P = 48$), however, the number of states visited by CNDFS ($|V|$) increases due to work duplication.

To further investigate the scalability of the timed CNDFS algorithm, we plot the speedups in Fig. 7. Vertical bars represent the (mostly negligible) standard deviation over the five benchmarks. Three benchmarks exhibit linear scalability, while train-gate and fddi show a sub-linear, yet still positive, trend. For train-gate, we suspect that this is caused by the structure of the state space. Because fddi has only 119 explicit state parts, we attribute the poor scalability to lock contention, harming more with a growing number of workers.

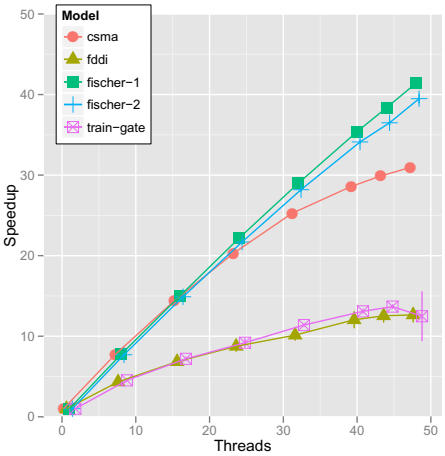


Fig. 7. Speedups in LTSMIN/opaal

Subsumption. Table 2 shows the experimental data for BFS, DFS and CNDFS with subsumption (Alg. 3). The number of explicit state parts $|L|$ is stable, since reachability of locations is preserved under subsumption (Prop. 2). However, the achieved reduction of full states depends on the search order, so we now report $|\mathcal{R}|$ per algorithm, as a percentage of the original numbers.

Table 2. Runtimes and states counts *with* subsumption (in % relative to Table 1)

Model	P	$\lvert\mathcal{R}\rvert_{bfs}$	$\lvert\mathcal{R}\rvert_{dfs}$	$\lvert\mathcal{R}\rvert_{cndfs}$	$\lvert V\rvert_{cndfs}$	$\lvert\Rightarrow\rvert_{bfs}$	T_{bfs}	T_{dfs}	T_{cndfs}
csma	1	48.7	88.9	58.3	94.7	41.2	41.3	90.3	95.2
csma	48	48.7	77.5	58.3	93.6	41.2	64.5	85.3	97.8
fddi	1	3.1	3.4	50.8	53.1	3.4	4.3	4.7	132.3
fddi	48	3.1	2.4	50.8	80.1	3.4	51.0	19.5	121.0
fischer-1	1	17.9	72.4	55.2	91.9	27.0	25.6	78.7	97.3
fischer-1	48	17.9	71.1	55.2	95.9	27.0	33.1	79.6	103.0
fischer-2	1	18.6	68.5	77.5	95.8	28.7	27.0	75.3	98.9
fischer-2	48	18.6	62.7	77.5	95.8	28.7	37.4	72.5	98.3
train-gate	1	100.0	100.0	100.0	100.0	100.0	100.6	100.6	104.3
train-gate	48	100.0	100.0	100.0	100.0	100.0	101.7	83.5	83.1

We confirm [3] that subsumption works best for BFS reachability, with even more than 30-fold reduction for fddi, but none for fischer (cf. column $|\mathcal{R}|_{bfs}$). For these benchmarks, the reduction is correlated to the ratio $X = |\mathcal{R}|/|L|$; e.g. $X \approx 1500$ for fddi and $X \approx 10$ for fischer. Subsumption is much less effective with sequential DFS, but parallel DFS improves it slightly (cf. column $|\mathcal{R}|_{dfs}$).

CNDFS benefits considerably from subsumption, but less so than BFS: we observe around 2-fold reduction for fddi, fischer and csma (cf. column $|\mathcal{R}|_{cndfs}$). Surprisingly, the reduction for parallel runs of CNDFS is not better than for sequential runs. One disadvantage of CNDFS compared to BFS is that only red states attribute to subsumption reduction. Probably some "large" states are never coloured red. We measured that for all benchmark models, 20%–50% of all reachable states are coloured red (except for fischer-2, which has no red states).

Subsumption decreases the running times for reachability: a lot for BFS, and still considerably for DFS, both in the sequential case and the parallel case, up to 48 workers. However, subsumption is less beneficial for the running time of CNDFS (it might even increase), but the speedup remains unaffected.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Baier, C., Katoen, J.-P.: Principles of Model Checking. The MIT Press (2008)
3. Behrmann, G.: Distributed reachability analysis in timed automata. STTT 7(1), 19–30 (2005)
4. Behrmann, G., Bouyer, P., Larsen, K.G., Pelánek, R.: Lower and upper bounds in zone based abstractions of timed automata. In: Jensen, K., Podelski, A. (eds.) TACAS 2004. LNCS, vol. 2988, pp. 312–326. Springer, Heidelberg (2004)
5. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
6. Bengtsson, J.: Clocks, DBMs and States in Timed Systems. PhD thesis, Uppsala University (2002)
7. Blom, S., van de Pol, J., Weber, M.: LTSMIN: Distributed and symbolic reachability. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 354–359. Springer, Heidelberg (2010)
8. Bouajjani, A., Tripakis, S., Yovine, S.: On-the-fly symbolic model checking for real-time systems. In: 18th IEEE RTSS, pp. 25–34. IEEE (1997)

9. Courcoubetis, C., Vardi, M.Y., Wolper, P., Yannakakis, M.: Memory efficient algorithms for the verification of temporal properties. In: Clarke, E., Kurshan, R.P. (eds.) CAV 1990. LNCS, vol. 531, pp. 233–242. Springer, Heidelberg (1991)
10. Dalsgaard, A.E., Hansen, R.R., Jørgensen, K.Y., Larsen, K.G., Olesen, M.C., Olsen, P., Srba, J.: `opaal`: A lattice model checker. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 487–493. Springer, Heidelberg (2011)
11. Dalsgaard, A.E., Laarman, A., Larsen, K.G., Olesen, M.C., van de Pol, J.: Multi-core reachability for timed automata. In: Jurdziński, M., Ničković, D. (eds.) FORMATS 2012. LNCS, vol. 7595, pp. 91–106. Springer, Heidelberg (2012)
12. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
13. Dill, D.L.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
14. Evangelista, S., Laarman, A., Petrucci, L., van de Pol, J.: Improved multi-core nested depth-first search. In: Chakraborty, S., Mukund, M. (eds.) ATVA 2012. LNCS, vol. 7561, pp. 269–283. Springer, Heidelberg (2012)
15. Evangelista, S., Petrucci, L., Youcef, S.: Parallel nested depth-first searches for LTL model checking. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 381–396. Springer, Heidelberg (2011)
16. Holzmann, G.J., Peled, D., Yannakakis, M.: On nested depth-first search. In: The Spin Verification System, 2nd SPIN Workshop, pp. 23–32. AMS (1996)
17. Laarman, A., Langerak, R., van de Pol, J., Weber, M., Wijs, A.: Multi-core nested depth-first search. In: Bultan, T., Hsiung, P.-A. (eds.) ATVA 2011. LNCS, vol. 6996, pp. 321–335. Springer, Heidelberg (2011)
18. Laarman, A.W., van de Pol, J.C.: Variations on multi-core nested depth-first search. In: PDMC, vol. 72, pp. 13–28 (2011)
19. Laarman, A.W., van de Pol, J.C., Weber, M.: Boosting multi-core reachability performance with shared hash tables. In: FMCAD. IEEE Computer Society (2010)
20. Laarman, A., van de Pol, J., Weber, M.: Multi-core lTSmin: Marrying modularity and scalability. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 506–511. Springer, Heidelberg (2011)
21. Laarman, A., van de Pol, J., Weber, M.: Parallel recursive state compression for free. In: Groce, A., Musuvathi, M. (eds.) SPIN Workshops 2011. LNCS, vol. 6823, pp. 38–56. Springer, Heidelberg (2011)
22. Larsen, K., Pettersson, P., Yi, W.: Uppaal in a nutshell. STTT 1, 134–152 (1997)
23. Li, G.: Checking timed büchi automata emptiness using LU-abstractions. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 228–242. Springer, Heidelberg (2009)
24. Schwoon, S., Esparza, J.: A note on on-the-fly verification algorithms. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 174–190. Springer, Heidelberg (2005)
25. Tripakis, S.: Checking timed Büchi automata emptiness on simulation graphs. TOCL 10(3), 15 (2009)
26. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed Büchi automata emptiness efficiently. Formal Methods in System Design 26(3), 267–292 (2005)
27. Vardi, M.Y., Wolper, P.: An automata-theoretic approach to automatic program verification. In: LICS, pp. 332–344. IEEE (1986)

PSyHCoS: Parameter Synthesis for Hierarchical Concurrent Real-Time Systems

Étienne André[1], Yang Liu[2], Jun Sun[3], Jin Song Dong[4], and Shang-Wei Lin[5,⋆]

[1] Université Paris 13, Sorbonne Paris Cité, LIPN, F-93430, Villetaneuse, France
[2] Nanyang Technological University, Singapore
[3] Singapore University of Technology and Design, Singapore
[4] School of Computing, National University of Singapore
[5] Temasek Laboratories, National University of Singapore

Abstract. Real-time systems are often hard to control, due to their complicated structures, quantitative time factors and even unknown delays. We present here PSyHCoS, a tool for analyzing parametric real-time systems specified using the hierarchical modeling language PSTCSP. PSyHCoS supports several algorithms for parameter synthesis and model checking, as well as state space reduction techniques. Its architecture favors reusability in terms of syntax, semantics, and algorithms. It comes with a friendly user interface that can be used to edit, simulate and verify PSTCSP models. Experiments show its efficiency and applicability.

1 Introduction

Ensuring the correctness of safety-critical systems, involving complex data structures with timing requirements, is crucial. The correctness of such real-time systems usually depends on the values of timing delays. Checking the correctness for one particular value for each delay is usually not sufficient for two reasons. Firstly, values for the delays are not always known, and one may precisely want to *find* some values for which the system behaves well. Secondly, even if the system is proved to be correct for a reference set of values, one has no guarantee that the correctness holds for other values around the reference ones. This is known as the *robustness* (see, e.g., [11]) of the system. Often, the engineer knows a correct reference valuation, but testing the correctness of the system for many values around it can turn very costly. Hence, it is interesting to consider the delays as unknown constants, or *parameters*, and synthesize constraints on these parameters to guarantee the correct behavior.

In this work, we present PSyHCoS (Parameter SYnthesis for Hierarchical COncurrent Systems), which supports editing, simulating, parameter synthesis and parametric model checking for Parametric Stateful Timed CSP (PSTCSP) [3]. PSyHCoS is a self-contained toolkit with extensible architecture design. To

⋆ This work is mainly supported by the TRF Project Grant No. R394-000-063-23 and the Seed Project Grant No. R394-000-068-232 from Temasek Lab@National University of Singapore.

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 984–989, 2013.

the best of our knowledge, PSyHCoS is the first tool that synthesizes timing parameters for real-time systems handling both hierarchy and concurrency.

The language PSTCSP offers an intuitive syntax for modeling hierarchical real-time systems involving shared variables, complex data structures, and user defined programs. PSTCSP (that is a parametric extension of Stateful Timed CSP [12]) is a process algebra with syntax for specifying concurrency (including conditional, general, external, internal choices, etc.) and timing requirements such as `Wait`$[d]$, `timeout`$[d]$, `within`$[d]$ and `deadline`$[d]$, where d can be a constant or a timing parameter. The expressiveness of PSTCSP is incomparable[1] with Parametric Timed Automata (PTA), which are an extension of finite state automata with clocks (variables increasing linearly) and parameters. Different from PTA, clocks in PSTCSP are *implicit* and dynamically created during the execution, thus avoiding the designer to write clocks constraints manually, which is error-prone. Another advantage of PSTCSP over PTA is the ability to easily define hierarchical systems, where sub-systems can be defined independently. Many systems can be designed more intuitively using hierarchy, and it may allow one to handle refinement as well as closed ("black box" or "gray box") systems.

Related Work. UPPAAL [10] is a tool for verifying (extensions of) timed automata. UPPAAL does not handle parameter synthesis and, although an extension performs parametric model checking [6], the model remains non-parametric. Some hierarchical extensions were considered, with limited tool support (e.g., [8,7]).

In [9], the process algebra ACSR-VP is used to synthesize timing constraints such that a real-time system is schedulable. PSyHCoS shares some principles with this approach, but does not limit to scheduling.

IMITATOR [2] is a tool performing parameter synthesis for PTA extended with stopwatches. Although IMITATOR has been extensively used, in particular for verifying models of industrial circuits, it does not feature any GUI nor simulation facilities and is limited to the inverse method (see Section 2). Last but not least, PSyHCoS can natively handle hierarchy whereas IMITATOR cannot.

2 Parameter Synthesis Made Easy

PSyHCoS offers a complete GUI for the design, simulation and verification of PSTCSP models. It comes with a user friendly editing environment (multi-document, multi-language interface, and advanced syntax editing features) for composing models. PSyHCoS also features a simulator that can be used for interactively and visually simulating system behaviors by random simulation, user-guided step-by-step simulation, complete state graph generation, trace playback, etc. Screenshots of the interface are available in PSyHCoS's Web page [1].

Among the verification algorithms, PSyHCoS first implements the inverse method *IM* initially defined for PTA [4] and extended to PSTCSP [3]. *IM* takes as input a PSTCSP model as well as a reference valuation π for all the parameters; it synthesizes a convex constraint K, that guarantees the same time-abstract behavior (sequences of actions) as for π. A major advantage is that K gives a

[1] Precisely, PSTCSP is equivalent to parametric closed timed ϵ-automata (see [3]).

quantitative measure of the robustness of the system w.r.t. variations of the timing delays. In particular, all linear time properties that hold for π also hold for any valuation in K. Parameter synthesis for PSTCSP has been proved to be undecidable [3] (as for PTA), and we were able to build examples on purpose for which *IM* does not terminate. However, for all practical case studies we considered, PSyHCoS does terminate. Exhibiting subclasses of PSTCSP for which termination of *IM* is guaranteed is the subject of ongoing work.

A full reachability algorithm *reachAll* is also implemented in PSyHCoS, in order to compare optimization techniques (see Table 1). Other classical model checking algorithms (such as LTL, deadlock freeness, or refinement checking) are also available. Data structures and functions can be written using the programming languages like C# and used seamlessly in the PSTCSP models.

We use Fischer's mutual exclusion algorithm to show PSyHCoS's intuitive modeling facilities. This hierarchical process (starting from `Fischer`) uses 2 timing parameters (`Delta` and `Epsilon`) and 2 variables. The `turn` variable indicates which process attempted to access the critical section most recently. The `counter` variable counts the number of processes accessing the critical section.

```
1  #define N 3;
2  #define Idle -1;
3  var turn = Idle;
4  var counter = 0;
5  parameter Delta;
6  parameter Epsilon;
7
8  proc(i) = ifb(turn == Idle) { Active(i) };
9  Active(i)=((update.i{turn=i} -> Wait[Epsilon])within[Delta]);
10    if (turn == i) {
11       cs.i{counter++} -> exit.i{counter--;turn=Idle}->proc(i)
12    } else {
13       proc(i)};
14 Fischer = ||| i:{0..N-1}@proc(i);
15
16 #synthesize Fischer with Delta = 3, Epsilon = 4;
```

`N` is a constant representing the number of processes. The parallel composition (line 14) automatically creates `N` processes in parallel. Process `proc(i)` models a process with a unique integer identify `i`. If `turn` is `Idle` (i.e., no other process is attempting), `proc(i)` behaves as specified by `Active(i)`. In `Active(i)`, `turn` is first set to `i` (i.e., the `i`th process is now attempting) by action `update.i`. Note that `update.i` must occur within `Delta` time units (captured by `within[Delta]`). Next, the process idles for `Epsilon` time units. It then checks if `turn` is still `i`. If so, it enters the critical section and leaves later. Otherwise, it restarts from the beginning.

A classical parameter synthesis problem is to find values for `Delta` and `Epsilon` such that mutual exclusion is guaranteed. This is achieved by calling the inverse method (at the last line) with `Delta=3` and `Epsilon=4` as a reference valuation.

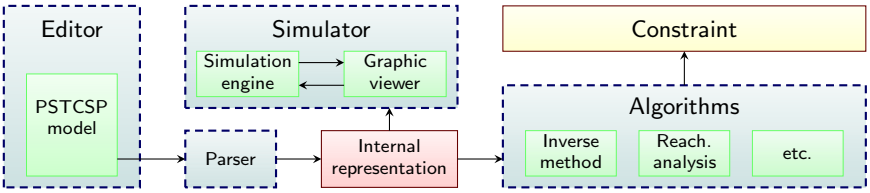


Fig. 1. Architecture of PSyHCoS

The constraint synthesized by PSyHCoS is `Delta` $<$ `Epsilon`, viz., the weakest (i.e., best) constraint known to guarantee mutual exclusion.

3 An Architecture Favoring Reusability

PSyHCoS is implemented in C#, based on Microsoft .NET framework, and uses the PPL library [5] for solving the parameter constraints. Sources, binaries, user manual and case studies are available in [1].

Reusability. The architecture of PSyHCoS is given in Fig. 1; it is fully object-oriented to favor reusability and ease the addition of new features. Each semantic rule of [3] is implemented in a different class, the methods of which are called every time the rule is applied. As a consequence, adding a new syntactic construct and its associated semantic rule simply requires one to add a new class to implement the semantics of the new syntax, and add a new line to the parser grammar file. Similarly, all algorithms are implemented in different classes. Although some algorithms are of course specific to PSTCSP, it is also possible to add algorithms that only depend on labeled transitions systems (such as LTL-checking, deadlock freeness, etc.). Such algorithms could be imported at no cost from existing model checking algorithms operating on LTS, e.g., Bogor, LTSA or the PAT model checking library.

Internal representation. The semantics of PSTCSP is defined as a labeled transition system, where the states in the transition system consist of a process and a constraint on clocks and parameters [3]. Each state is implemented as a pair (process id, constraint id), both coded as a string. Although some processing is needed each time a new state is computed, the constraint equality test reduces to string equality, which is faster than other representations. Furthermore, a string format is flexible – which is interesting as we are dealing with hierarchical systems so that different states may have very different system architecture.

Optimization. PSyHCoS implements a state-space reduction technique, that merges equivalent states in PSTCSP [3]. In the best case, this leads to an exponential diminution of the number of states, but at the cost of several nontrivial operations. The optimized version of *reachAll* (resp. *IM*) is denoted by *reachAll+* (resp. *IM+*). Both approaches are implemented so that users can choose.

Table 1. Application of algorithms for parameter synthesis using PSyHCoS

Case study	$\|U\|$	*reachAll* $\|S\|$	$\|T\|$	$\|X\|$	t	*reachAll+* $\|S\|$	$\|T\|$	$\|X\|$	t	*IM* $\|S\|$	$\|X\|$	t	*IM+* $\|S\|$	$\|X\|$	t
Bridge	4	-	-	-	M.O.	-	-	-	M.O.	2.8k	2	253	2.8k	2	455
Fischer_4	2	-	-	-	M.O.	-	-	-	M.O.	11k	4	41.9	2k	4	8.65
Fischer_5	2	-	-	-	M.O.	-	-	-	M.O.	133k	5	1176	13k	5	84.5
Fischer_6	2	-	-	-	M.O.	-	-	-	M.O.	-	-	M.O.	86k	6	1144
Jobshop	8	14k	20k	2	21.0	12k	17k	2	18.1	1112	2	17.1	877	2	22.8
RCS_5	4	5.6k	7.2k	4	10.5	5.6k	7.2k	4	9.54	5.6k	4	7.83	5.6k	4	16.7
RCS_6	4	34k	43k	4	91.7	34k	43k	4	54.5	34k	4	60.4	34k	4	91.3
TrAHV	6	7.2k	13k	6	14.2	7.2k	13k	6	15.8	227	6	0.555	227	6	0.655

4 Experiments and Discussion

We applied PSyHCoS to synthesize parameters for real-time systems ranging from classical concurrent algorithms to real world problems. In Table 1, we list the example names, the number $|U|$ of parameters and, for each algorithm, the number $|S|$ (resp. $|T|$) of states (resp. transitions), the maximum number $|X|$ of clocks, and the computation time t in seconds on a Windows XP 32 bits computer with an Intel Quad Core 2.4 GHz processor and 4 GB memory.

Bridge is a bridge crossing problem for 4 persons within 17 time units. Fischer_i is the mutual exclusion protocol for i processes. Jobshop is a scheduling problem. TrAHV is a classical train example for PTA. RCS_i is a railway control system with i trains. The reference valuation used for *IM* either is the standard valuation for the considered problem (Bridge, Jobshop, RCS_i, TrAHV) or has been computed in order to satisfy a well-known constraint of good behavior (Fischer_i).

When *reachAll* terminates, we can apply classical model checking algorithms: e.g., we checked that all models are deadlock-free (except Jobshop[2]). When *reachAll* does not terminate (Bridge, Fischer), *IM* is interesting because it synthesizes constraints despite an infinite set of reachable states; and when *reachAll* terminates slowly (TrAHV), *IM* may synthesize constraints much quicker.

The constraint output has several advantages. Firstly, it synthesizes values for which the system behaves well. Secondly, it gives a criterion of robustness to the system, by defining a safety domain around each parameter. Thirdly, it can happen that the constraint is *True* (e.g., RCS_i for all i). In this case, one can safely *refine* the model by removing all timing constructs (`Wait`, `deadline`, etc.). Although this might be checked using refinement techniques for one particular parameter valuation, we prove it here for *any* parameter valuation.

Also observe that, when *IM+* indeed reduces the number of states, it is much more efficient than *IM*, not only w.r.t. memory, but also w.r.t. time. However, with no surprise, when no state duplication is met (e.g., Bridge), the computation time is greater. Although reducing this computation is a subject of ongoing work, we do not consider it as a significant drawback: parameter synthesis' largest

[2] Jobshop is an acylic scheduling problem, where tasks should execute only once. Hence, once all actions have been performed, the system ends with a deadlock.

limitations are usually non-termination and memory saturation. Slower analyses for some case studies (up to +80% for Bridge) are acceptable when others benefit from a dramatic memory (and time) reduction (-90% for Fischer_5), allowing parameter synthesis even when *IM* goes out of memory (Fischer_6).

Comparison with IMITATOR. A comparison with IMITATOR (using the same machine with Ubuntu 11.10 64 bits) turned inaccurate. Indeed, the (manual) translation of models from PTSCP to PTA (and conversely) is difficult: in all cases, the tool for which the model was initially designed performs much better than the tool that runs on a translated model. For example, Jobshop (8.96 s) and TrAHV (0.097 s) are quicker on IMITATOR, for which they are designed. Conversely, IMITATOR does not terminate for Fischer_i for all i because of the explicit clock representation in PTA, whereas the implicit clocks in PSTCSP prevent this. Other models (Bridge, RCS_i) are too large to be manually translated. An automated efficient translation mechanism, that could ease such a comparison, is a subject of future work. Nevertheless, some features specific to PSTCSP, such as hierarchy, data structures, and implicit clocks, would be lost in any case by the translation.

References

1. PSyHCoS page, `http://lipn.univ-paris13.fr/~andre/software/PSyHCoS/`
2. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012)
3. André, É., Liu, Y., Sun, J., Dong, J.S.: Parameter synthesis for hierarchical concurrent real-time systems. In: ICECCS 2012, pp. 253–262 (2012)
4. André, É., Soulat, R.: The Inverse Method. ISTE Ltd and John Wiley & Sons Inc. (2013)
5. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library: Toward a complete set of numerical abstractions for the analysis and verification of hardware and software systems. Science of Computer Programming 72(1-2), 3–21 (2008)
6. Behrmann, G., Larsen, K.G., Rasmussen, J.I.: Beyond liveness: Efficient parameter synthesis for time bounded liveness. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 81–94. Springer, Heidelberg (2005)
7. David, A., Larsen, K.G., Legay, A., Nyman, U., Wąsowski, A.: ECDAR: An environment for compositional design and analysis of real time systems. In: Bouajjani, A., Chin, W.-N. (eds.) ATVA 2010. LNCS, vol. 6252, pp. 365–370. Springer, Heidelberg (2010)
8. Dong, J.S., Hao, P., Qin, S., Sun, J., Yi, W.: Timed automata patterns. IEEE Transactions on Software Engineering 34(6), 844–859 (2008)
9. Kwak, H.-H., Lee, I., Sokolsky, O.: Parametric approach to the specification and analysis of real-time system designs based on ACSR-VP. Electronic Notes in Theoretical Computer Science 25, 38–49 (1999)
10. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer 1(1-2), 134–152 (1997)
11. Markey, N.: Robustness in real-time systems. In: SIES 2011, pp. 28–34. IEEE Computer Society Press (2011)
12. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., André, É.: Modeling and verifying hierarchical real-time systems using Stateful Timed CSP. ACM Transactions on Software Engineering and Methodology 22(1), 3.1–3.29 (2013)

Lazy Abstractions for Timed Automata*

Frédéric Herbreteau[1], B. Srivathsan[2], and Igor Walukiewicz[1]

[1] Univ. Bordeaux, CNRS, LaBRI, UMR 5800, F-33400 Talence, France
[2] Software Modeling and Verification Group, RWTH Aachen University, Germany

Abstract. We consider the reachability problem for timed automata. A standard solution to this problem involves computing a search tree whose nodes are abstractions of zones. For efficiency reasons, they are parametrized by the maximal lower and upper bounds (LU-bounds) occurring in the guards of the automaton. We propose an algorithm that dynamically updates LU-bounds during exploration of the search tree. In order to keep them as small as possible, the bounds are refined only when they enable a transition that is impossible in the unabstracted system. So our algorithm can be seen as a kind of lazy CEGAR algorithm for timed automata. We show that on several standard benchmarks, the algorithm is capable of keeping very small LU-bounds, and in consequence is able to reduce the search space substantially.

1 Introduction

Timed automata are obtained from finite automata by adding clocks that can be reset and whose values can be compared with constants. The reachability problem asks if a given target state is reachable from the initial state by an execution of the automaton. The standard solution to this problem computes the so-called zone graph of the automaton, and uses abstractions to make the algorithm both terminating and more efficient.

Most abstractions are based on constants used in comparisons of clock values. Such abstractions have already been considered in the seminal paper of Alur and Dill [1]. Behrmann et. al. [4] have proposed abstractions based on LU-bounds, that are two functions L and U: the L function assigns to every clock a maximal constant appearing in a lower bound constraint in the automaton; similarly U associates the maximum constant appearing in an upper bound constraint. In a recent paper [15] we have shown how to efficiently use the $\mathfrak{a}_{\preccurlyeq LU}$ abstraction of [4] that is parameterized by LU-bounds. Moreover, $\mathfrak{a}_{\preccurlyeq LU}$ has been proved to be the biggest abstraction that is sound for all automata with given LU-bounds. Since $\mathfrak{a}_{\preccurlyeq LU}$ abstraction of a zone can result in a non-convex set, we have shown in [15] how to use this abstraction without the need to store the result of the abstraction. This opens new algorithmic possibilities because changing LU-bounds becomes very cheap as abstractions need not be recalculated. In this paper we explore these possibilities.

* This work was partially supported by the ANR project Vacsim (ANR-11-INSE-004).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 990–1005, 2013.

The algorithm we propose works as follows. It constructs a graph with nodes of the form (q, Z, LU), where q is a state of the automaton, Z is a zone, and LU are parameters for the abstraction. It starts with the biggest abstraction: LU bounds are set to $-\infty$ which makes $\mathfrak{a}_{\preccurlyeq LU}(Z)$ to be the set of all valuations for every nonempty Z. The algorithm explores the zone graph using standard transition relation on zones, without modifying LU bounds till it encounters a disabled transition. More concretely, till it reaches a node (q, Z, LU) such that there is a transition from q that is not possible from (q, Z) because no valuation in Z allows to take it. At this point we need to adjust LU bounds so that the transition is not possible from $\mathfrak{a}_{\preccurlyeq LU}(Z)$ either. This adjustment is then propagated backwards through the already constructed part of the graph.

The real challenge is to limit the propagation of bound updates. For this, if the bounds have changed in a node $(q', Z', L'U')$ then we consider its predecessor nodes (q, Z, LU) and update its LU bounds as a function of Z, Z' and $L'U'$. We give general conditions for correctness of such an update, and a concrete efficient algorithm implementing it. This requires getting into a careful analysis of the influence of the transition on the zone Z. As a result we obtain an algorithm that exhibits exponential gains on some standard benchmarks.

We have analyzed the performance of our algorithm theoretically as well as empirically. We have compared it with a static analysis algorithm, namely the state-of-the-art algorithm implemented in UPPAAL, and with an algorithm we have proposed in [14]. The latter improves on the static analysis algorithm by considering only the reachable part of the zone graph. For an example borrowed from [17] we have proved that the algorithm presented here produces a linear size search graph while for the other two algorithms, the search graph is exponential in the size of the model. For the classic FDDI benchmark, which has been tested on just about every algorithm for the reachability problem, our algorithm shows the rather surprising fact that the time component is almost irrelevant. There is only one constraint that induces LU bounds, and in consequence the abstract search graph constructed by our algorithm is linear in the size of the parameter of FDDI.

Our algorithm can be seen as a kind of CEGAR algorithm similar in spirit to [13], but then there are also major differences. In the particular setting of timed automata the information available is much richer, and we need to use it in order to obtain a competitive algorithm. First, we do not need to wait till a whole path is constructed to analyze if it is spurious or not. Once we decide to keep zones in nodes we can immediately detect if an abstraction is too large: it is when it permits a transition not permitted from the zone itself. Next, the abstractions we use are highly specialized for the reachability problem. Finally, the propagation of bound changes gets quite sophisticated because it can profit from the large amount of useful information in the exploration graph.

Related work. Forward analysis is the main approach for the reachability testing of real-time systems. The use of zone-based abstractions for termination has been introduced in [10]. The notion of LU-bounds and inference of these bounds by static analysis of an automaton have been proposed in [3,4]. The $\mathfrak{a}_{\preccurlyeq LU}$

approximation has been introduced in [4]. An approximation method based on LU-bounds, called $Extra^+_{LU}$, is used in the current implementation of UPPAAL [5]. In [15] we have shown how to efficiently use $\mathfrak{a}_{\preccurlyeq_{LU}}$ approximation. We have also proposed an LU-propagation algorithm [14] that can be seen as applying the static analysis from [3] on the zone graph instead of the graph of the automaton; moreover this inference is done on-the-fly during construction of the zone graph. In the present paper we do much finer inference and propagation of LU-bounds.

Approximation schemes for the analysis of timed-automata have been considered almost immediately after the introduction of the concept of timed automata, as for example in [2,22,12] or [20]. All these papers share the same idea to abstract the region graph by not considering all the constraints involved in the definition of a region. When a spurious counterexample is discovered a new constraint is added. So in the worst case the whole region graph will be constructed. Our algorithm in the worst case constructs an $\mathfrak{a}_{\preccurlyeq_{LU}}$-abstracted zone graph with LU-bounds obtained by static analysis. This is as good as the state-of-the-art method used in UPPAAL. Another slightly related paper is [7] where the CEGAR approach is used to handle diagonal constraints.

Let us mention that abstractions are not needed in the backward exploration of timed systems. Nevertheless, any feasible backward analysis approach needs to simplify constraints. For example [19] does not use approximations and relies on an SMT solver instead. This approach, or the approach of RED [21], are very difficult to compare with the forward analysis approach we study here.

Organization of the paper. In the preliminaries section we introduce all standard notions we will need, and $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction in particular. Section 3 gives a definition of adaptive simulation graph (ASG). Such a graph represents the search space of a forward reachability testing algorithm that will search for an abstract run with respect to $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, while changing LU-bounds dynamically during exploration. Section 4 gives an algorithm for constructing an ASG with small LU-bounds. Section 5 presents the two crucial functions used in the algorithm: the one updating the bounds due to disabled edges, and the one propagating the change of bounds. Section 6 explains some advantages of our algorithm on variations of an example borrowed from [17]. The experiments section compares our prototype tool with UPPAAL, and our algorithm from [14]. The conclusions section gives some justification for our choice of concentrating on LU-bounds. The proofs are presented in the full version of the paper [16].

2 Preliminaries

2.1 Timed Automata, Zones, and the Reachability Problem

Let X be a set of clocks, i.e., variables that range over $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. A *guard* is a conjunction of constraints $x\#c$ for $x \in X$, $\# \in \{<, \leq, =, \geq, >\}$ and $c \in \mathbb{N}$, e.g. $(x \leq 3 \wedge y > 0)$. Let $\Phi(X)$ denote the set of clock constraints over clock variables X. A *clock valuation* over X is a function

$v : X \to \mathbb{R}_{\geq 0}$. We denote by $\mathbf{0}$ the valuation that associates 0 to every clock in X. We write $v \vDash \phi$ when v satisfies the guard ϕ. For $\delta \in \mathbb{R}_{\geq 0}$, let $v + \delta$ be the valuation that associates $v(x) + \delta$ to every clock x. For $R \subseteq X$, let $v[R]$ be the valuation that sets x to 0 if $x \in R$, and that sets x to $v(x)$ otherwise.

A *timed automaton (TA)* is a tuple $\mathcal{A} = (Q, q_0, X, T, Acc)$ where Q is a finite set of states, $q_0 \in Q$ is the initial state, X is a finite set of clocks, $Acc \subseteq Q$ is a set of accepting states, and $T \subseteq Q \times \Phi(X) \times 2^X \times Q$ is a finite set of transitions (q, g, R, q') where g is a *guard*, and R is the set of clocks that are *reset* on the transition.

A *configuration* of $\mathcal{A}$ is a pair $(q, v) \in Q \times \mathbb{R}_{\geq 0}^X$ and $(q_0, \mathbf{0})$ is the *initial configuration*. We have two kinds of transitions:

Delay: $(q, v) \to^{\delta} (q, v + \delta)$ for some $\delta \in \mathbb{R}_{\geq 0}$;

Action: $(q, v) \to^{t} (q, v[R])$ for a transition $t = (q, g, R, q') \in T$ such that $v \vDash g$.

A *run* of $\mathcal{A}$ is a finite sequence of transitions starting from the initial configuration $(q_0, \mathbf{0})$. A run is *accepting* if it ends in a configuration (q_n, v_n) with $q_n \in Acc$.

Definition 1 (Reachability problem). *The* reachability problem *for timed automata is to decide whether there exists an accepting run of a given automaton.*

This problem is known to be PSPACE-complete [1,9]. The class of TA we consider is usually known as diagonal-free TA since clock comparisons like $x - y \leq 1$ are disallowed. Notice that if we are interested in state reachability, considering timed automata without state invariants does not entail any loss of generality as the invariants can be added to the guards. For state reachability, we can also consider automata without transition labels.

Rather than working with valuations we will work with sets of valuations and symbolic transitions. So we will consider configurations of the form (q, W) where q is a state of the automaton and W a set of valuations.

Definition 2 (Symbolic transition $\Rightarrow$). *Let $\mathcal{A}$ be a timed automaton. For every transition t of $\mathcal{A}$ and every set of valuations W, we define:*

$$(q, W) \Rightarrow^{t} (q', W') \text{ where } W' = \{v' \mid \exists v \in W,\ \exists \delta \in \mathbb{R}_{\geq 0}.\ (q, v) \to^{t} \to^{\delta} (q', v')\}$$

We will sometimes write $\mathrm{Post}_t(W)$ *for W'. The transition relation $\Rightarrow$ is the union of all $\Rightarrow^t$.*

Observe that symbolic transitions always yield sets closed under time-successors. Such sets are called *time-elapsed*. Let W_0 be the set of valuations reachable from the the initial valuation $\mathbf{0}$ by a time elapse: $W_0 = \{\mathbf{0} + \delta \mid \delta \in \mathbb{R}_{\geq 0}\}$.

A *symbolic run* is a sequence of symbolic transitions:

$$(q_0, W_0) \Rightarrow (q_1, W_1) \Rightarrow \dots$$

It has been noticed that the sets W appearing in a symbolic run can be described by some simple constraints involving only the difference between clocks [6]. This has motivated the definition of *zones*, which are sets of valuations defined by difference constraints.

Definition 3 (Zones [6]). *A zone is a set of valuations defined by a conjunction of two kinds of clock constraints: $x \sim c$ and $x - y \sim c$ for $x, y \in X$, $c \in \mathbb{Z}$, and $\sim \in \{\leq, <, =, >, \geq\}$.*

Observe that W_0 is a zone: it is given by the constraints $\bigwedge_{x,y \in X} (x \geq 0 \wedge x - y = 0)$. Then every W appearing in a symbolic run is a zone too. It is well-known that an automaton has an accepting run if and only if it has a symbolic run reaching an accepting state and non-empty zone. As the number of zones that can appear in a symbolic run is not bounded, the next simplification step is to consider abstractions of zones.

2.2 LU-bounds and LU-abstractions

The most common parameter taken for defining abstractions are LU-bounds.

Definition 4 (LU-bounds). *The L bound for an automaton $\mathcal{A}$ is the function assigning to every clock x a maximal constant that appears in a lower bound guard for x in $\mathcal{A}$, that is, the maximum over guards of the form $x > c$ or $x \geq c$. Similarly U is the function assigning to every clock x a maximal constant appearing in an upper bound guard for x in $\mathcal{A}$, that is, the maximum over guards of the form $x < c$ or $x \leq c$. If there is no guard in question then the value of $L(x)$ or $U(x)$ is $-\infty$.*

The paper introducing LU-bounds [4] also introduced an abstraction operator $\mathfrak{a}_{\preccurlyeq_{LU}}$ that uses LU-bounds as parameters. We begin by recalling the definition of an LU-preorder defined in [4]. We use a different but equivalent formulation.

Definition 5 (LU-preorder and $\mathfrak{a}_{\preccurlyeq_{LU}}$-abstraction [4]). *Let $L, U : X \to \mathbb{N} \cup \{-\infty\}$ be two bound functions. For a pair of valuations we set $v \preccurlyeq_{LU} v'$ if for every clock x:*

- *if $v'(x) < v(x)$ then $v'(x) > L_x$, and*
- *if $v'(x) > v(x)$ then $v(x) > U_x$.*

For a set of valuations W we define:

$$\mathfrak{a}_{\preccurlyeq_{LU}}(W) = \{v \mid \exists v' \in W.\ v \preccurlyeq_{LU} v'\}.$$

An efficient algorithm to use the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction for reachability was proposed in [15]. Moreover in op. cit. it was shown that over time-elapsed zones, $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is optimal when the only information about the analyzed automaton are its LU-bounds. Informally speaking, for a fixed LU, the $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction is the biggest abstraction that is sound and complete for all automata using guards within LU-bounds.

Since the abstraction $\mathfrak{a}_{\preccurlyeq_{LU}}$ is optimal, the next improvement is to try to get as good LU-bounds as possible since tighter bounds give coarser abstractions, and in consequence induce a smaller search space.

It has been proposed in [3] that instead of considering one LU-bound for all states in an automaton, one can use different bound functions for each state. For

every state q and every clock x, constants $L_x(q)$ and $U_x(q)$ are determined by the least solution of the following set of inequalities. For each transition (q, g, R, q') in the automaton, we have:

$$\begin{cases} L_x(q) \geq c & \text{if } x \geq c \text{ or } x > c \text{ is a constraint in } g \\ L_x(q) \geq L_x(q') & \text{if } x \notin R \end{cases} \tag{1}$$

Similar inequalities are written for U, now considering $x \leq c$ and $x < c$. It has been shown in [3] that such an assignment of constants is sound and complete for state reachability. Experimental results have shown that this method, that performs a static analysis on the automaton, often gives very big gains.

3 Adaptive Simulation Graph

In this paper we improve on the idea of static analysis that computes LU-bounds for each state q. We will compute LU-bounds on-the-fly for each node (q, Z) of the zone graph, while simultaneously searching for an accepting run. The key difference is that the bounds will depend not only on the state but also on the set of valuations. This will immediately allow us to ignore guards from unreachable parts of the automaton. However, the real freedom given by an adaptive simulation graph and Theorem 1 presented below is that when calculating the LU-bounds, they will allow to ignore some guards of transitions even from the reachable part. As we will see in the experiments section, this can result in significant gains.

We will construct a forward reachability algorithm that will search for an abstract run with respect to $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, where the LU-bounds change dynamically during exploration. The intuition of a search space of such an algorithm is formalized in a notion of an adaptive simulation graph (ASG). Such a graph permits to change LU-bounds from node to node, provided some consistency conditions are satisfied. This is important as LU-bounds are used to stop exploring successors of a node. So our goal will be to find as small LU-bounds as possible in order to stop developing the graph as soon as possible.

Definition 6 (Adaptive simulation graph (ASG)). *Fix an automaton $\mathcal{A}$. An ASG has nodes of the form (q, Z, LU) where q is the state of $\mathcal{A}$, Z is a zone, and LU are bound functions. Some nodes are declared to be tentative. The graph is required to satisfy three conditions:*

G1 *For the initial state q_0 and initial zone Z_0, a node (q_0, Z_0, LU) should appear in the graph for some LU.*

G2 *If a node (q, Z, LU) is not tentative then for every transition $(q, Z) \Rightarrow_t (q', Z')$ the node should have a successor labeled $(q', Z', L'U')$ for some $L'U'$.*

G3 *If a node (q, Z, LU) is tentative then there should be a non-tentative node $(q', Z', L'U')$ such that $q = q'$ and $Z \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z')$. Node $(q', Z', L'U')$ is called covering node.*

We will also require that the following invariants are satisfied:

I1 *If a transition* $\Rightarrow^t$ *is disabled from* (q, Z)*, and* (q, Z, LU) *is a node of the ASG then* $\Rightarrow^t$ *should be disabled from* $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ *too;*

I2 $\mathrm{Post}_t(\mathfrak{a}_{\preccurlyeq_{LU}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq_{L'U'}}(Z')$*, for every edge* $(q, Z, LU) \Rightarrow_t (q', Z', L'U')$ *of the ASG.*

I3 $L_2U_2 \leq L_1U_1$*, for every tentative node* (q, Z_1, L_1U_1) *and the corresponding covering node* (q, Z_2, L_2U_2)*.*

In the above $L_2U_2 \leq L_1U_1$ says that for all clocks x, $L_2(x) \leq L_1(x)$ and $U_2(x) \leq U_1(x)$. The conditions G1, G2, G3 express the expected requirements for a graph to cover all reachable configurations. In particular, the condition G3 allows to stop the exploration if there is already a "better" node in the graph. The three invariants are more subtle. They imply that LU-bounds should be big enough for the reachability information to be preserved. (cf. Theorem 1).

Remark: While the idea is to work with nodes of the form (q, W) with W being a result of $\mathfrak{a}_{\preccurlyeq_{LU}}$ abstraction, we do not want to store W directly, as we have no efficient way of representing and manipulating such potentially non-convex sets. Instead we represent each W as $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$. So we store Z and LU. This choice is algorithmically cheap since testing the inclusion $Z' \subseteq \mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ is practically as easy as testing $Z' \subseteq Z$ [15]. This approach has another big advantage: when we change the LU bound in a node, we do not need to recalculate $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$.

Remark: It is important to observe that for every $\mathcal{A}$ there exists a finite ASG. For this it is sufficient to take static LU-bounds as described by (1). It means that we can take the ASG whose nodes are $(q, Z, L(q)U(q))$ with bound functions given by static analysis [3]. It is easy to see that such a choice makes all three invariants hold.

The next theorem tells us that any ASG is good enough to determine the existence of an accepting run. Our objective in the following section will be to construct an ASG as small as possible.

Theorem 1. *Let* G *be an ASG for an automaton* $\mathcal{A}$*. An accepting state is reachable by a run of* $\mathcal{A}$ *iff a node containing an accepting state of* $\mathcal{A}$ *and a non-empty zone is reachable from the initial node of* G*.*

4 Algorithm

In this section, we present an algorithm that computes an ASG satisfying conditions G1, G2, G3 and maintaining invariants I1, I2, I3 of Definition 6. The basic algorithm is the same as in [14]. The LU-bounds are calculated dynamically while constructing the ASG. The main difference lies in the way LU-bounds are calculated so that the ASG is small, but still maintains the invariants.

The algorithm constructs the ASG in a form of a tree with cross edges from tentative nodes. The nodes v in this tree consist of four components: $v.q$ is a state of $\mathcal{A}$, $v.Z$ is a zone, and $v.L$, $v.U$ are LU bound functions. Each node v has a successor v_t for every transition t of $\mathcal{A}$ that results in a non-empty zone

from $v.Z$. Some nodes will be marked tentative and not explored further. After an exploration phase, tentative nodes will be reexamined and some of them will be put on the stack for further exploration. At every point the leaves of the tree constructed by the algorithm will be of three kinds: tentative nodes, nodes on the stack, nodes having no transition to be explored.

The exploration proceeds by a standard depth-first search. When a node v is called for exploration, we assume that the values $v.q$ and $v.Z$ are set. Moreover, $v.Z$ must be non-empty. The initial values of $v.L$ and $v.U$ are set to $-\infty$. We also assume that the constructed tree satisfies the invariants I1, I2, I3, except for the node v and the nodes on the stack. If the state $v.q$ is accepting then we have found an accepting run and the algorithm terminates reporting "not empty". Otherwise, the procedure needs to explore the successors of v and restore invariants, if needed. For this, it is first checked if there exists a non-tentative node v' in the tree such that $v.Z \subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(v'.Z)$. If it is true, then v is marked tentative wrt v' and $v.LU$ is set to $v'.LU$ in order to maintain I3. The node v is not explored further. If such a non-tentative node cannot be found in the tree, the successors of v are computed and put on the stack. To ensure I1, a function `disabled` is called that gives new bounds for $v.LU$. We explain this function in the next section.

When LU-bounds in some node v are changed, the invariant I2 should be restored for its ancestors. For this, the modified bounds are propagated upward along the tree. The parent v_p of v is taken and the transition from v_p to v is examined. A function `newbounds` is called on v_p. This function calculates new LU bounds for a node given the changes in its successor, so that I2 is ensured. This function is the core of our algorithm and is the subject of the next section. If the bounds of v_p indeed change then they should be copied to all nodes tentative w.r.t. v_p. This is necessary to satisfy the invariant I3. Finally the bounds are propagated to the predecessor of v_p to restore invariant I2.

An exploration phase stops if there are no more nodes in the stack. During the course of the exploration, the LU bounds of tentative nodes might have changed. A procedure `resolve` is called to check for the consistency of tentative nodes. If v is tentative w.r.t. v' but $v.Z \not\subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(v'.Z)$ is not true anymore, v needs to be explored. Hence it is viewed as a new node, marked non-tentative, and put on the stack for further exploration.

The algorithm terminates when either it finds an accepting state, or there are no nodes to be explored and all tentative nodes remain tentative. In the second case we can conclude that the constructed tree represents an ASG, and hence no accepting state is reachable. Note that the overall algorithm should terminate as the bounds can only increase and bounds in a node (q, Z) are not bigger than the bounds obtained for q by static analysis (cf. Remark on page 996). The correctness of the algorithm then follows from Proposition 1.

Proposition 1. *The algorithm always terminates. If for a given $\mathcal{A}$ the result is "not empty" then $\mathcal{A}$ has an accepting run. Otherwise the algorithm returns empty after constructing ASG for $\mathcal{A}$ and not seeing an accepting state.*

5 Controlling *LU*-bounds

In the previous section, we described the basic algorithm to construct an ASG with LU-bounds at each node. We had not addressed the issue of how to calculate these LU-bounds. In this section we describe the two functions used by the algorithm to calculate LU-bounds: `disabled` and `newbounds`.

The notion of adaptive simulation graph (Definition 6) gives necessary conditions for the values of LU bounds in every node. The invariant I1 tells that LU bounds in a node should take into account the edges disabled from the node. The invariant I2 gives a lower bound on LU with respect to the LU-bounds in successors of the node. Finally, I3 tells us that LU bounds in a covered node should not be smaller than in the covering node. The task of the functions `disabled` and `newbounds` is precisely to maintain the three invariants without increasing the bounds unnecessarily. The pseudo-code of these functions is presented in [16].

Proviso: For simplicity, we assume a special form of transitions of timed automata. A transition can have either only upper bound constraints, or only lower bound constraints and no resets. Observe that a transition $q_1 \xrightarrow{g;R} q_2$ is equivalent to $q_1 \xrightarrow{g_L} q_1' \xrightarrow{g_U;R} q_2$; where g_L is the conjunction of the lower bound guards from g and g_U is the conjunction of the upper bound guards from g. So in order to satisfy our proviso we may need to double the number of states of an automaton.

The `disabled` function is quite simple. Its task is to restore the invariant I1. For this the function chooses from every disabled transition an atomic guard that makes it disabled. Recall that we have assumed that every guard contains either only lower bound constraints or only upper bound constraints. A transition with only lower bound constraints cannot be disabled from a time elapsed zone. Hence a guard on a disabled transition must be a conjunction of upper bound constraints. It can be shown that if such a guard is not satisfied in a zone then there is one atomic constraint of the guard that is not satisfied in a zone. Now it suffices to use Definition 5 and observe that if a constraint $x \leq d$ or $x < d$ is not satisfied in Z then it is not satisfied in $\mathfrak{a}_{\preccurlyeq_{LU}}(Z)$ when $U(x) \geq d$.

In the rest of this section we describe the function $\texttt{newbounds}(v, v', X'_L, X'_U)$. This function calculates new LU-bounds for v, given that the bounds in v' have changed. As an additional information we use the sets of clocks X'_L and X'_U that have changed their L-bound, and U-bound respectively, in v'. This information makes the `newbounds` function more efficient since the new bounds depend only on the clocks in X'_L and X'_U. The aim is to give bounds that are as small as possible and at the same time satisfy invariant I2 from Definition 6.

For space reasons we will consider only the case when a guard is a single constraint and there is no reset. The extension to a conjunction of constraints does not pose particular problems. Treating reset is easy. We treat transitions having guard and reset at the same time as a combination of purely guard and purely reset transitions. We refer the reader to the full paper for details [16].

We consider a transition $(q, Z, LU) \Rightarrow_g (q', Z', L'U')$, that is a transition with a guard but no reset. Suppose that we have updated $L'U'$ and now we want our `newbounds` function to compute $L_{new}U_{new}$. In the constant propagation algorithm of [14], we would have set $L_{new}U_{new}$ to be the maximum over LU, $L'U'$, and the constant present in the guard. This is sufficient to maintain Invariant 2. However, it is not necessary to always take the guard g into consideration for the propagation.

Let L_gU_g be the bound function induced by the guard g. In our case, as there is only one constraint, there is only one constant associated to a single clock by L_gU_g. It can be shown that in order to maintain Invariant 2, it suffices to take

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U') & \text{if } [\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \text{ or} \\ & \text{if } Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z') \\ \max(LU, L'U', L_gU_g) & \text{otherwise} \end{cases} \tag{2}$$

Since bound propagation is called often in the main algorithm, we need an efficient test for the inclusions in formula (2). The formula requires us to test inclusion w.r.t. $\mathfrak{a}_{\preccurlyeq LU}$ between Z and Z' each time we want to calculate $L_{new}U_{new}$. Although this seems complicated at the first glance, note that Z' is a zone obtained by a successor computation from Z. When we have only a guard in the transition, we have $Z' = \overrightarrow{Z \wedge g}$: in words, zone Z' is obtained by first intersecting Z with g and letting time elapse from the resulting set of valuations. This relation between Z and Z' makes the inclusion test a lot more simpler. We will also see that it is not necessary to consider the inclusion $[\![g]\!] \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

Before proceeding, we are obliged to look closer at how zones are represented. Instead of difference bound matrices (DBMs) [11], we will prefer an equivalent representation in terms of distance graphs.

A *distance graph* has clocks as vertices, with an additional special clock x_0 representing the constant 0. For readability, we will often write 0 instead of x_0. Between every two vertices there is an edge with a weight of the form $(\prec, c)$ where $c \in \mathbb{Z}$ and $\prec$ is either $\leq$ or $<$; or $(\prec, c)$ equals $(<, \infty)$. An edge $x \xrightarrow{\prec c} y$ represents a constraint $y - x \prec c$: or in words, the distance from x to y is bounded by c. A distance graph is in *canonical form* if the weight of the edge from x to y is the lower bound of the weights of paths from x to y. A zone Z can be identified with the distance graph in the canonical form representing the constraints in Z. For two clocks x, y we write Z_{xy} for the weight of the edge from x to y in this graph. A special case is when x or y is 0, so for example Z_{0y} denotes the weight of the edge from 0 to y.

We recall a theorem from [15] that yields an efficient test for: $Z \subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$.

Theorem 2. *Let Z, Z' be two non-empty zones. Then $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ iff there exist two clocks x, y such that:*

$$Z_{x0} \geq (\leq, -U'_x) \text{ and } Z'_{xy} < Z_{xy} \text{ and } Z'_{xy} + (<, -L'_y) < Z_{x0} \tag{3}$$

We are ready to proceed with our analysis.

Lower bound guard: When the guard on the transition is $w \geqslant d$, the diagonals, that is Z_{xy} with both x, y variables other than zero, do not change during intersection and time-elapse. Hence we have $Z'_{xy} = Z_{xy}$ for such x and y. This shows that (3) cannot be true when both x and y are non-zero as the second condition is false. Yet again, when x is 0, the second condition cannot be true since both $Z_{0y} = Z'_{0y} = (<, \infty)$, as after time-elapse there is no constraint of the form $y \lessdot c$ where $c \in \mathbb{Z}$. It remains to consider the single case when y is 0. It boils down to checking if there exists a clock x such that $Z_{x0} \geq (\leq, -U'_x)$ and $Z'_{x0} < Z_{x0}$. In words, the test asks if there exists a clock x whose label of the edge $x \xrightarrow{\lessdot -c} 0$ in Z has reduced in Z' and additionally the edge weight $(\lessdot, -c)$ in Z satisfies either $c < U'_x$ or $(\lessdot, c) = (\leq, U'_x)$. It can be checked that if $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, then $[\![g]\!] \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$ too. So in this case Formula (3) simplifies to the following formula with the additional observation that Z'_{x0} can be only lesser than or equal to Z_{x0}.

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } \exists x.\ \big(Z_{x0} \geq (\leq, -U'_x)\big) \wedge \big((Z'_{x0} < Z_{x0})\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases}$$

Note that this test can be easily extended to an incremental procedure: whenever we modify the U' value of a clock, we need to check only this clock. The above definition also suggests that whenever only L' is modified we don't have to check anything and just propagate the new values of L'.

Upper bound guard: When we have an upper bound guard, the diagonals might change. However no edge $0 \to x$ or $x \to 0$ changes. Therefore we need to check (3) for two non-zero variables x and y.

In other words, among clocks x that have a finite U' constant and clocks y that have a finite L' constant, we check if there is a diagonal $x \to y$ that has strictly reduced in Z' and additionally satisfies $Z'_{xy} + (<, L_y) < Z_{x0}$. Yet again, it can be checked that if $Z \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$, then $[\![g]\!] \not\subseteq \mathfrak{a}_{\preccurlyeq L'U'}(Z')$. Therefore it is sufficient to check (3) for non zero variables x and y. This gives the following formula function:

$$L_{new}U_{new} = \begin{cases} \max(LU, L'U', L_gU_g) & \text{if } \exists x, y. \text{ such that} \\ & \big(Z_{x0} \geq (\leq, -U'_x)\big) \ \wedge \ \big(Z'_{xy} < Z_{xy}\big) \wedge \\ & \quad \big(Z'_{xy} + (<, -L'_y) < Z_{x0}\big) \\ \max(LU, L'U') & \text{otherwise} \end{cases} \tag{4}$$

This test can also be done incrementally. Each time we propagate, we need to perform extra checks only when a new clock has got a finite value for either L' or U'.

Other types of transitions with upper bound guard and resets, and multiple guards are treated in the full version of the paper [16]. The pseudocode of the `newbounds` function implementing the obtained tests is presented in the full

version too. Here, we just state the theorem expressing the correctness of the construction.

Theorem 3. *Let v' be a node of ASG and v be its parent. Let $(v.q, v.Z) \Rightarrow_t (v'.q, v'.Z')$ be a transition. Given bound functions $v'.LU$, the functions $L_{new}U_{new}$ as computed by* `newbounds`(v, v', X'_L, X'_U) *satisfy invariant I2, namely:*

$$\mathrm{Post}_t(\mathfrak{a}_{\preccurlyeq L_{new}U_{new}}(Z)) \subseteq \mathfrak{a}_{\preccurlyeq v'.LU}(Z').$$

6 Examples

In this section we will analyze the behavior of our algorithm on some examples in order to explain some of the sources of the gains reported in the experiments of the next section.

The invariants in the definition of adaptive simulation graph (Definition 6) sometimes allow for much smaller LU-bounds than that obtained by static analysis. A very basic example is when during exploration the algorithm does not encounter a node with a disabled edge. In this case all LU-bounds are simply $-\infty$, since propagation does not change such bounds. If LU bounds are $-\infty$, and Z is nonempty then $\mathfrak{a}_{\preccurlyeq LU}(Z)$ is the set of all valuations. So in this case the ASG is just a subgraph of the automaton. We will now see an example where such a situation occurs and yields exponential gain over the static analysis method used by UPPAAL, and the on-the-fly constant propagation algorithm from [14].

Consider the automaton $\mathcal{D}_n$ shown in Figure 1. This is a slightly modified example from [17]. We have changed all guards to check for an equality. Automaton $\mathcal{D}_n$ is a parallel composition of three components. The first two components respectively reset the x-clocks and y-clocks. The third component can be fired only after the first two have reached their a_n states. The reachable states of the product automaton $\mathcal{D}_n$ are of the form (a_i, a_j, b_0) and (a_n, a_n, b_k) where $i, j, k \in \{0, \ldots, n\}$. Let us assume that no state is accepting so that any forward exploration algorithm should explore the entire search space.

Clearly, all the transitions can be fired if no time elapses in the states (a_i, a_j, b_0) for $i, j \in 1, \ldots, n-1$, and exactly one time unit elapses in (a_n, a_n, b_0). Therefore, an ASG for $\mathcal{D}_n$ will have no edges disabled which implies that in each node the LU-constants given by our algorithm are $-\infty$. The number of uncovered nodes in the ASG constructed by our algorithm will be the same as the number of states.

However, the static analysis procedure would give $L = U = 1$ for every clock. It can be proved that this would yield a zone graph with at least 2^n nodes. As all the edges are enabled, the constant propagation algorithm from [14] would explore a path up to (a_n, a_n, b_n). This would therefore give $L = U = 1$ for each clock, similar to static analysis. So in this case too there would be at least 2^n uncovered nodes in the reachability tree obtained.

Let us now see an example when there is a disabled edge. Consider the automaton $\mathcal{A}_2$ in Figure 1. One can see that the last transition with the upper bound is not fireable, and that the reason is the guard $x \geq 5$. Our algorithm

would say that at q_0 the relevant constants are: $L_0(x) = 5$ and $U_0(x) = 1$ and the rest are $-\infty$. The static analysis algorithm or the constant propagation would give additionally $L(y) = 5$ and $L(z) = 100$, which is unnecessary. We will now see that this pruning can sometimes lead to an exponential gain.

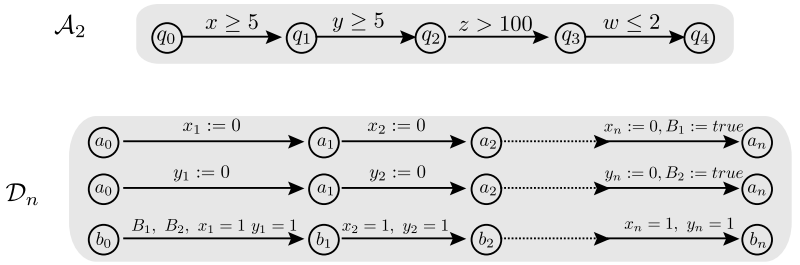


Fig. 1. Automata $\mathcal{A}_2$ and $\mathcal{D}_n$

Consider automaton $\mathcal{D}'_n$ obtained from $\mathcal{D}_n$ in Figure 1 by changing every constraint involving the y-clock to $y = 2$. If the algorithm is "fortunate" to choose the right order of resets, it can reach node (a_n, a_n, b_n) without seeing a disabled edge. Due to this it will construct an ASG with the number of uncovered nodes equal to number of states of the automaton.

If it is not the case, then it reaches the state (a_n, a_n, b_0) with a zone where there is an i such that $y_i \leq x_i$ and for all $j < i$, $x_i \leq y_i$. From such a zone, the path can be taken till b_{i-1} after which the transition gets disabled because we check for $y_i \geq 2$ and $x_i \leq 1$. The disabled edge gives the constant $U(x_i) = 1$ and the propagation algorithm additionally generates $L(y_i) = 2$. But it generates no bounds for other clocks. In the result, the reached node will cover any other node $((a_n, a_n, b_0), Z', L'U')$ with Z' satisfying $x_i \leq y_i$. So there will be at most n uncovered nodes with the state (a_n, a_n, b_0) and hence the total number of uncovered nodes will be at most quadratic in n. In fact, it will be linear but for this a more careful analysis is needed.

The static analysis procedure would give $L = U = 2$ for all y-clocks and $L = U = 1$ for all x-clocks. It can be shown that this would result in at least 2^n uncovered nodes with state (a_n, a_n, b_0).

The on-the-fly propagation algorithm [14] could work slightly differently from the previous case. The constants generated depend on the first path. If the first path leads up to (a_n, a_n, b_n) then there are constants generated for all clocks. Then, the zone cannot cover any of the future zones that appear at (a_n, a_n, b_n). A depth-first search algorithm would clearly then be exponential. Otherwise, if the path gets cut at b_{k-1} constants are generated for all clocks $x_1, y_1, \ldots, x_k, y_k$. In this case, at least 2^k nodes at (a_n, a_n, b_0) need to be distinguished.

7 Experiments

We report experiments in Table 1 for classical benchmarks from the literature. The first two columns compare UPPAAL 4.1.13 with our own implementation

Table 1. Comparison of reachability algorithms: number of visited nodes and running time. For each model and each algorithm, we kept the best of depth-first search and breadth-first search. Experiments done on a MacBook with 2.4GHz Intel Core Duo processor and 2GB of memory running MacOS X 10.6.8. Missing numbers are due to time out (150s) or memory out (1Gb).

Model	nb. of	UPPAAL (-C)		$Extra^+_{LU}$,sa		$\mathfrak{a}_{\preccurlyeq LU}$,otf		$\mathfrak{a}_{\preccurlyeq LU}$,disabled	
	clocks	nodes	sec.	nodes	sec.	nodes	sec.	nodes	sec.
$\mathcal{D}''_7$	14	18654	11.6	18654	8.1	213	0.0	72	0.0
$\mathcal{D}''_8$	16					274	0.0	90	0.0
$\mathcal{D}''_{70}$	140							5112	1.9
CSMA/CD 10	11	120845	1.9	120844	6.3	78604	6.1	51210	4.0
CSMA/CD 11	12	311310	5.4	311309	16.8	198669	16.1	123915	10.2
CSMA/CD 12	13	786447	14.8	786446	44.0	493582	41.8	294924	25.2
FDDI 50	151	12605	52.9	12606	29.4	5448	14.7	401	0.8
FDDI 70	211							561	2.7
FDDI 140	421							1121	40.6
Fischer 9	9	135485	2.4	135485	8.9	135485	11.4	135485	14.8
Fischer 10	10	447598	10.1	447598	34.0	447598	42.8	447598	56.8
Fischer 11	11	1464971	40.4	1464971	126.8				
Stari 2	7	7870	0.1	6993	0.4	5779	0.4	4305	0.4
Stari 3	10	136632	1.7	113958	9.4	82182	8.2	43269	4.5
Stari 4	13	1323193	26.2	983593	109.0	602762	84.9	296982	41.5

of UPPAAL's algorithm ($Extra^+_{LU}$,sa). We have taken particular care to ensure that the two implementations deal with the same model and explore it in the same way. However, on the last example (Stari), we did not manage to force the same search order in the two tools.

The last two algorithms are using bounds propagation. In the third column ($\mathfrak{a}_{\preccurlyeq LU}$,otf), we report results for the algorithm in [14] that propagates the bounds from every transition (enabled or disabled) that is encountered during the exploration of the zone graph. Since this algorithm only considers the bounds that are reachable in the zone graph, it generally visits less nodes than UPPAAL's algorithm. The last column ($\mathfrak{a}_{\preccurlyeq LU}$,disabled) corresponds to the algorithm introduced in this paper. It propagates the bounds that come from the disabled transitions only. As a result it generally outperforms the other algorithms. The actual implementation of our algorithm is slightly more sophisticated than the one presented in Section 4. Like UPPAAL, it uses a Passed/Waiting list instead of a stack. The implemented algorithm is presented in the Appendix of [16].

The results show a huge gain on two examples: $\mathcal{D}''$ and FDDI. D''_n corresponds to the automaton $\mathcal{D}_n$ in Fig. 1 where the tests $x_k = 1, y_k = 1$ have been replaced by $(0 < x_k \leq 1), (1 < y_k \leq 2)$. While it was easier in Section 6 to analyze the example with equality tests, we wanted here to show that the same performance gain occurs also when static L bounds are different from static U bounds. The number of nodes visited by algorithm $\mathfrak{a}_{\preccurlyeq LU}$,disabled exactly corresponds to the number of states in the timed automaton. The situation with the FDDI example is similar: it has only one disabled transition. The other three algorithms take useless clock bounds into account. As a result they quickly face a combinatorial explosion in the number of visited nodes. We managed to analyze $\mathcal{D}''_n$ up to $n = 70$ and FDDI up to size 140 despite the huge number of clocks.

Fischer example represents the worst case scenario for our algorithm. Dynamic bounds calculated by algorithms $\mathfrak{a}_{\preccurlyeq_{LU}}$,otf and $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled turn out to be the same LU-bounds given by static analysis.

The remaining two models, CSMA/CD and Stari [8] show the average situation. The interest of Stari is that it is a very complex example with both a big discrete part and a big continuous part. The model is exactly the one presented in op. cit. but for a fixed initial state. Algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled discards many clock bounds by considering disabled transitions only. This leads to a significant gain in the number of visited nodes at a reasonable cost.

8 Conclusions

We have pursued an idea of adapting abstractions while searching through the reachability space of a timed automaton. Our objective has been to obtain as low LU-bounds as possible without sacrificing practicability of the approach. In the end, the experimental results show that algorithm $\mathfrak{a}_{\preccurlyeq_{LU}}$,disabled improves substantially the state-of-the art forward exploration algorithms for the reachability problem in timed automata.

At first sight, a more refined approach would be to work with constraints themselves instead of LU-abstractions. Following the pattern presented here, when encountering a disabled transition, one could take a constraint that makes it disabled, and then propagate this constraint backwards using, say, weakest precondition operation. A major obstacle in implementing this approach is the covering condition, like G3 in our case. When a node is covered, a loop is formed in the abstract system. To ensure soundness, the abstraction in a covered node should be an invariant of this loop. A way out of this problem can be to consider a different covering condition as proposed by McMillan [18], but then this condition requires to develop the abstract model much more than we do. So from this perspective we can see that LU-bounds are a very interesting tool to get a loop invariant cheaply, and offer a good balance between expressivity and algorithmic effectiveness.

We do not make any claim about optimality of our backward propagation algorithm. For example, one can see that it gives different results depending on the order of treating the constraints. Even for a single constraint, our algorithm is not optimal in a sense that there are examples when we could obtain smaller LU-bounds. At present we do not know if it is possible to compute optimal LU-bounds efficiently. In our opinion though, it will be even more interesting to look at ways of cleverly rearranging transitions of an automaton to limit bounds propagation even further. Another promising improvement is to introduce some partial order techniques, like parallelized interleaving from [19]. We think that the propagation mechanisms presented here are well adapted to such methods.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. TCS 126(2), 183–235 (1994)
2. Alur, R., Itai, A., Kurshan, R.P., Yannakakis, M.: Timing verification by successive approximation. Inf. Comput. 118(1), 142–157 (1995)

3. Behrmann, G., Bouyer, P., Fleury, E., Larsen, K.G.: Static guard analysis in timed automata verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 254–270. Springer, Heidelberg (2003)
4. Behrmann, G., Bouyer, P., Larsen, K.G., Pelanek, R.: Lower and upper bounds in zone-based abstractions of timed automata. Int. Journal on Software Tools for Technology Transfer 8(3), 204–215 (2006)
5. Behrmann, G., David, A., Larsen, K.G., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST, pp. 125–126. IEEE Computer Society (2006)
6. Bengtsson, J.E., Yi, W.: Timed automata: Semantics, algorithms and tools. In: Desel, J., Reisig, W., Rozenberg, G. (eds.) ACPN 2003. LNCS, vol. 3098, pp. 87–124. Springer, Heidelberg (2004)
7. Bouyer, P., Laroussinie, F., Reynier, P.-A.: Diagonal constraints in timed automata: Forward analysis of timed systems. In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 112–126. Springer, Heidelberg (2005)
8. Bozga, M., Maler, O., Tripakis, S.: Efficient verification of timed automata using dense and discrete time semantics. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 125–141. Springer, Heidelberg (1999)
9. Courcoubetis, C., Yannakakis, M.: Minimum and maximum delay problems in real-time systems. Form. Methods Syst. Des. 1(4), 385–415 (1992)
10. Daws, C., Tripakis, S.: Model checking of real-time reachability properties using abstractions. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 313–329. Springer, Heidelberg (1998)
11. Dill, D.: Timing assumptions and verification of finite-state concurrent systems. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407, pp. 197–212. Springer, Heidelberg (1990)
12. Dill, D.L., Wong-Toi, H.: Verification of real-time systems by successive over and under approximation. In: Wolper, P. (ed.) CAV 1995. LNCS, vol. 939, pp. 409–422. Springer, Heidelberg (1995)
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)
14. Herbreteau, F., Kini, D., Srivathsan, B., Walukiewicz, I.: Using non-convex approximations for efficient analysis of timed automata. In: FSTTCS. LIPIcs, vol. 13, pp. 78–89 (2011)
15. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Better abstractions for timed automata. In: LICS, pp. 375–384 (2012)
16. Herbreteau, F., Srivathsan, B., Walukiewicz, I.: Lazy abstractions for timed automata. arXiv:1301.3127, Extended version with proofs (2013)
17. Lugiez, D., Niebert, P., Zennou, S.: A partial order semantics approach to the clock explosion problem of timed automata. TCS 345(1), 27–59 (2005)
18. McMillan, K.L.: Lazy abstraction with interpolants. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 123–136. Springer, Heidelberg (2006)
19. Morbé, G., Pigorsch, F., Scholl, C.: Fully symbolic model checking for timed automata. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 616–632. Springer, Heidelberg (2011)
20. Sorea, M.: Lazy approximation for dense real-time systems. In: Lakhnech, Y., Yovine, S. (eds.) FORMATS 2004 and FTRTFT 2004. LNCS, vol. 3253, pp. 363–378. Springer, Heidelberg (2004)
21. Wang, F.: Efficient verification of timed automata with BDD-like data structures. Int. J. Softw. Tools Technol. Transf. 6(1), 77–97 (2004)
22. Wong-Toi, H.: Symbolic Approximations of Verifying Real-Time Systems. PhD thesis, Stanford University (March 1995)

Shrinktech: A Tool for the Robustness Analysis of Timed Automata*

Ocan Sankur

LSV, ENS Cachan & CNRS, France
sankur@lsv.ens-cachan.fr

Abstract. We present a tool for the robustness analysis of timed automata that can check whether a given time-abstract behaviour of a timed automaton is still present when the guards are perturbed. The perturbation model we consider is *shrinking*, which corresponds to increasing lower bounds and decreasing upper bounds in the clock guards by parameters. The tool synthesizes these parameters for which the given behaviour is preserved in the new automaton if possible, and generates a counter-example otherwise. This can be used for 1) robustness analysis, and for 2) deriving implementations under imprecisions.

1 Introduction

Timed Automata and Robustness. Timed automata [3] are a well-established formal model for real-time systems. They can be used to model systems as finite automata, while using, in addition, a finite number of clocks to impose timing constraints on the transitions. Timed automata are, however, abstract models, and therefore make unrealistic assumptions on timings, such as perfect continuity of clocks, infinite-precision time measures and instantaneous reaction times. An important amount of work has been done in the timed automata literature to endow timed automata with a realistic semantics. The works [14] and [10] showed that perturbations on clocks, either imprecisions or clock drifts, and regardless of how small they are, may yield additional qualitative behaviours in some timed systems. On the other hand, assuming bounds on the reaction times can disable desired behaviours [8,1]. These observations mean that there is a need for checking the *robustness* of timed automata models, that is, whether the behaviour of a given timed automaton is preserved in presence of small perturbations. Robustness is an important property of critical embedded systems [12], since it requires that the system will behave correctly when the environment's behaviour deviates slightly from the assumptions.

Clock Imprecisions and Shrinking. A prominent approach to model imprecisions in timed automata, initiated in [11], consists in introducing imprecisions in the model by syntactically *enlarging* all guards, that is, turning a guard $x \in [a, b]$ into $x \in [a - \Delta, b + \Delta]$ for some parameter $\Delta > 0$. Model-checking algorithms on

* This work has been partly supported by project ImpRo (ANR-10-BLAN-0317).

N. Sharygina and H. Veith (Eds.): CAV 2013, LNCS 8044, pp. 1006–1012, 2013.

timed automata have been revisited in order to take into account such imprecisions (see *e.g.* [10,6]). These algorithms check whether any really new behaviour appears when timing constraints are relaxed by a small (parameterized) amount.

Recently we studied the dual notion of robustness, which consists in checking whether any behaviour is lost when the guards are *shrunk*, that is tightened by a small (parameterized) amount. More precisely, shrinking means converting a guard $x \in [a, b]$ into $x \in [a+\delta, b-\delta']$ for some $\delta, \delta' > 0$. In [15], we showed that one can decide whether all guards can be shrunk –by possibly different amounts, so that the resulting timed automaton can still time-abstract simulate the original automaton. In this case, one can also synthesize these shrinking parameters for each atomic guard. By checking shrinkability of timed automata, one ensures that the behaviour of the automaton does not depend on exact timings, or on its ability to take the transitions on the boundaries of the guards. A shrinkable timed automaton preserves all its behaviours when, for instance, task execution times are shorter than the worst-case, and waiting times are longer than the best-case. One can also detect unrealistic runs, including Zeno runs [15]. We believe that shrinkability complements the robustness approach based on guard enlargement of [11,10].

Shrinkability can also be used for deriving implementations with imprecise clocks. In fact, if the guard $x \in [a, b]$ is shrunk into $x \in [a + \delta, b - \delta]$, then under imprecisions modelled by guard enlargement (as in [11]), this guard becomes $x \in [a + \delta - \Delta, b - \delta + \Delta] \subseteq [a, b]$, where the inclusion holds whenever $\Delta < \delta$. Hence, the behaviours of a shrunk timed automaton with bounded imprecisions (*i.e.* guard enlargement) are entirely included in the behaviours of the initial timed automaton. Further, using shrinkability, one can synthesize parameters δ for each guard, so that the resulting automaton still contains some useful time-abstract behaviour.

Related Work. Existing verification tools for timed automata may be used for *non-parameterized* robustness checking by modeling explicitly the imprecisions, although this increases the size of the models [2]. The semi-algorithm of HyTech was used to synthesize guard enlargement parameters in timed automata in [11]. An extension of Uppaal for robustness against guard enlargement was used in [13]; this feature is no longer available in Uppaal. Note that shrinkability cannot be solved by existing model-checkers for timed automata since we are interested in parameter synthesis so as to ensure time-abstract simulation. Other similar work includes (the undecidable problem of) parameter synthesis in timed automata, where guards are written using parameters, and one tries to find the valuations for which the system satisfies some specification, *e.g.* [4]. This is difficult to realize due to the large number of parameters (upto millions) and the time-abstract simulation condition we consider. Robustness against large decreases in task execution times using simulation was considered in [1].

2 Shrinkability

Let us define shrinkability more formally. We assume that the reader is familiar with the syntax and semantics of Alur-Dill timed automata, and refer to [3] for

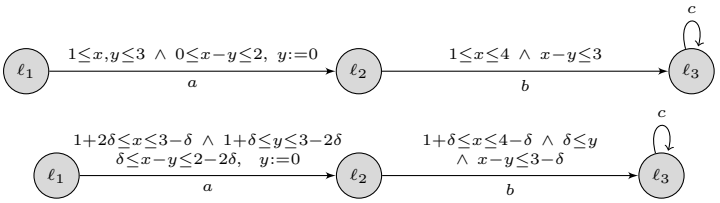


Fig. 1. A timed automaton $\mathcal{A}$ (above) and its shrinking $\mathcal{A}_{-\mathbf{k}\delta}$ (below). Timed automaton $\mathcal{A}_{-\mathbf{k}\delta}$ can time-abstract simulate $\mathcal{A}$ for all $\delta \in [0, \frac{1}{6}]$ ([15]).

details. We only need the following definitions. Given a finite clock set $\mathcal{C}$, an *atomic guard* is an expression of the form $x \le k \mid x \ge k \mid x - y \le k$, where $x, y \in \mathcal{C}$ and $k \in \mathbb{Z}$. A *guard* is a conjunction of atomic guards. The *shrinking* of an atomic guard g by δ, denoted by $\langle g \rangle_{-\delta}$ is defined as $\langle x \le k \rangle_{-\delta} = x \le k - \delta$, $\langle x \ge k \rangle_{-\delta} = x \ge k + \delta$, and $\langle x - y \le k \rangle_{-\delta} = x - y \le k - \delta$.

Note that the only variables that appear in timed automata are clocks. Discrete variables with bounded domains can be considered, but we assume these are encoded in the locations.

Let $\mathcal{A}$ be a timed automaton, and let I be the vector of the atomic guards of $\mathcal{A}$. Given a vector $\boldsymbol{\delta}$ of nonnegative rational numbers indexed by I, we denote by $\mathcal{A}_{-\boldsymbol{\delta}}$, the automaton obtained from $\mathcal{A}$ by shrinking each atomic guard by the corresponding element of $\boldsymbol{\delta}$. We are going to write the vector $\boldsymbol{\delta}$ as $\boldsymbol{k}\delta$ for an integer vector $\boldsymbol{k}$ and rational δ. This is always possible since we are interested in rational parameters. Figure 1 is an example of shrinking.

We are interested in shrinking the atomic guards of a given timed automaton by positive values, while preserving *some* of the behaviours. We only consider timed automata with non-strict guards; in fact, using strict guards makes little sense when one is interested in shrinking (or enlarging) the guards [10]. We also assume that the edges have distinct labels, since we are interested in comparing two timed automata that have the same underlying structure. The problem is formulated as follows:

Definition 1 (Shrinkability). *Given a timed automaton $\mathcal{A}$, and a finite automaton $\mathcal{F}$ such that $\mathcal{F} \sqsubseteq_{ta} \mathcal{A}$, decide whether for some $\boldsymbol{\delta} > 0$, $\mathcal{F} \sqsubseteq_{ta} \mathcal{A}_{-\boldsymbol{\delta}}$.*

In this definition, $\sqsubseteq_{\mathrm{ta}}$ denotes *time-abstract delay simulation*. We say that $\mathcal{A}$ is *shrinkable w.r.t.* $\mathcal{F}$ if the above condition is satisfied. Thus, shrinkability requires that some behaviour $\mathcal{F}$, that is included in the initial model $\mathcal{A}$, should be still possible in the shrunk automaton. When $\mathcal{F}$ is the region graph of $\mathcal{A}$ [3], or a time-abstract bisimulation quotient [16], shrinkability implies that the shrunk automaton can time-abstractly simulate the original automaton. In this case, we say that $\mathcal{A}$ is simply *shrinkable*. Shrinkability w.r.t. $\mathcal{F}$ is decidable in polynomial time if $\mathcal{F}$ is part of the input; shrinkability is decidable in exponential time, if the time-abstract bisimulation graph is not given. The vector $\boldsymbol{\delta}$ can be computed in the same time complexity [15].

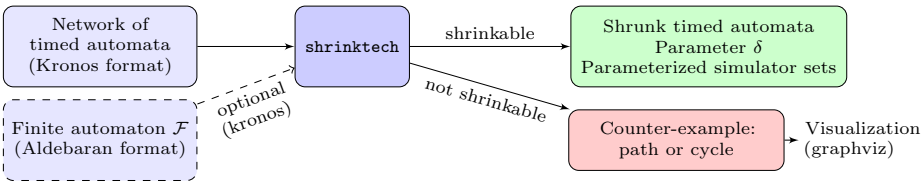


Fig. 2. Overview of `shrinktech`

The present tool builds on the theoretical results presented in [15]. There, we show that given a finite automaton $\mathcal{F}$, the shrinking parameter of each atomic guard can be expressed as a function of the other parameters using only maximization and sum. The problem is then reduced, in polynomial time, to solving nonlinear fixpoint equations in the max-plus algebra. We gave in [15] graph-based algorithms to solve these equations.

Partial shrinkability. Although we defined shrinkability by requiring that *all* atomic guards should be shrunk by a positive amount, one can relax this condition and shrink only some of the guards; our algorithms are valid also in this case. In our experiments, we shrunk all the guards but equality constraints.

3 The Tool `shrinktech`

We present the tool `shrinktech` that analyzes the shrinkability of timed automata and synthesizes shrinking parameters. Given a network of timed automata, the tool either finds a counter-example to shrinkability, such as a path or a cycle that cannot be executed by any shrinking of the automaton, whatever the value of δ's are, or outputs a shrinking of the timed automata that witnesses the shrinkability.

Figure 3 shows an overview of the tool. To check the shrinkability of a timed automaton, the user can either provide a finite automaton $\mathcal{F}$, or let `shrinktech` compute the full finite bisimilarity graph using Kronos[1]. Note that if the full bisimilarity graph is too big, one can also try to shrink with respect to a portion of it, or with respect to a randomly generated trace. This is to be compared with bounded model-checking, which is useful for detecting bugs, but also for "partially" proving the correctness of a system. The tool comes with scripts to compute the bisimilarity graph, extract some (random) portion of it, and generate random executions.

The tool `shrinktech` can be used for several kinds of systems modelled by timed automata. We believe it can be used mainly for two purposes:

1. Robustness analysis, to find out whether the behaviour of the system is preserved when the time bounds are disturbed (shrunk). This analysis complements the robustness checking by enlarging the guards as in [11,2]. This

[1] Kronos is a model-checker for timed automata [7] that can *minimize* the region graph of a timed automaton as described in [16]. It is available at `http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/kronos/`

can for instance help to detect unrealistic executions such as Zeno or other convergence phenomena, but also timing anomalies in scheduling problems (see [5]).
2. Deriving implementations from timed automata. As explained above, the behaviour of a shrunk timed automaton is included in that of the initial model in presence of imprecisions. So lower and upper bounds on the delays can be "shrunk" in the implementation to guarantee that these will be respected despite imprecisions. In fact, we proved in [15] that shrinkable timed automata can be implemented in a concrete semantics with imprecise clocks and reaction times.

Implementation details and availability. The tool is implemented in C++ and the source code has about 5Klocs. It uses the Uppaal DBM library[2], and implements a parameterized extension of this data structure, introduced in [15]. The input formats are (networks of) timed automata in the Kronos format, and finite automata in the Aldebaran format[3]. The tool Kronos can be plugged in the tool-chain in order to compute the finite time-abstract bisimilarity graph of a given timed automaton, to be used as the finite automaton $\mathcal{F}$. `Shrinktech` is open source software and is distributed under GNU General Public Licence 3.0. It is freely available at: `http://www.lsv.ens-cachan.fr/Software/shrinktech`

4 Experimental Results

We used `shrinktech` on several case studies found in the literature. The table 1 summarizes the results. The Lip Synchronization Protocol has been the subject of robustness analysis (by guard enlargement) before [13]. This is an algorithm that synchronizes video and sound streams that arrive in different frequencies. The model is not shrinkable neither for video frames arriving in exact frequency, nor for those arriving within a bounded interval. Observe that the model is shrinkable w.r.t. a small subgraph with 501 nodes, but it is not shrinkable w.r.t. the whole graph, which has 4484 nodes. Shrinkable models include Philips Audio Retransmission protocol [9], and some asynchronous circuit models. We were able to analyze Fischer's Mutual Exclusion Protocol upto 4 agents; while for 5 agents we could only partially analyze w.r.t. a randomly generated trace. The non-shrinkability of most models is due to equality constraints. In fact, although we only shrink non-punctual guards, some behaviours may still disappear immediately, however small the shrinking parameter is.

Note that some of these models were designed at a level of abstraction where imprecisions were not taken into account. So, our results do not necessarily imply that these systems are not robust, but rather that the present models are not good for direct implementation. This is best illustrated in the Latch Circuit models, where the exact model that extensively uses equalities is not shrinkable,

[2] `http://people.cs.aau.dk/~adavid/UDBM/`

[3] This is a graph description format of the CADP tool suite, also used by Kronos. See `http://www.inrialpes.fr/vasy/cadp/`

Table 1. The column sim-graph is the number of states and the number of transitions of the finite automaton $\mathcal{F}$ w.r.t. which the shrinkability is checked. An asterisk indicates bounded shrinkability, where only a subgraph of the time-abstract bisimulation graph (given by a BFS) or a random trace was used. The tests were performed on an Intel Xeon 2.67 GHz. All models are available on the tool's website.

Model	states	trans	clocks	sim-graph	time	shrinkable
Lip-Sync Prot. (Exact)	230	680	5	4000/8350* (subgraph)	9s	No
Lip-Sync Prot. (Interval)	230	680	5	501/1282* (subgraph)	9s	Yes*
Lip-Sync Prot. (Interval)	230	680	5	4484/48049	28s	No
Philips Audio Prot.	446	2097	2	437/2734	46s	Yes
Root Contention Prot.	65	138	6	500/3455* (subgraph)	7s	No
Train Gate Controller	68	199	11	952/8540	34s	No
Fischer's Protocol 3	152	464	3	472/4321	20s	Yes
Fischer's Protocol 4	752	2864	4	4382/65821	310min	Yes
Fischer's Protocol 5	3552	16192	5	10000/10000* (trace)	42s	Yes*
And-Or Circuit	12	20	4	80/497	1.3s	Yes
Flip-Flop Circuit	22	34	5	30/64	0.9s	Yes
Latch Circuit (Interval)	32	77	7	105/364	1.6s	Yes
Latch Circuit (Exact)	32	77	7	100/331	0.6s	No

but its relaxation to intervals is. Notice also how most of the circuit models which define bounds on stabilization times are shrinkable.

Our approach depends on the computation of the finite automaton $\mathcal{F}$, thus it is limited by the feasibility of this computation. To deal with this problem, one could consider adapting the algorithm of [16] to compute on-the-fly some bounded part of the graph. To be able to treat even larger systems, we will consider extending the theoretical results of [15], in order to use under- and over-approximations of the automaton $\mathcal{F}$, and refine these by counter-examples.

References

1. Abdellatif, T., Combaz, J., Sifakis, J.: Model-based implementation of real-time applications. In: EMSOFT 2010, pp. 229–238. ACM (2010)
2. Altisen, K., Tripakis, S.: Implementation of Timed Automata: An Issue of Semantics or Modeling? In: Pettersson, P., Yi, W. (eds.) FORMATS 2005. LNCS, vol. 3829, pp. 273–288. Springer, Heidelberg (2005)
3. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
4. André, É., Fribourg, L., Kühne, U., Soulat, R.: IMITATOR 2.5: A tool for analyzing robustness in scheduling problems. In: Giannakopoulou, D., Méry, D. (eds.) FM 2012. LNCS, vol. 7436, pp. 33–36. Springer, Heidelberg (2012)
5. Bouyer, P., Markey, N., Sankur, O.: Robust Reachability in Timed Automata: A Game-Based Approach. In: Czumaj, A., Mehlhorn, K., Pitts, A., Wattenhofer, R. (eds.) ICALP 2012, Part II. LNCS, vol. 7392, pp. 128–140. Springer, Heidelberg (2012)
6. Bouyer, P., Markey, N., Sankur, O.: Robust model-checking of timed automata via pumping in channel machines. In: Fahrenberg, U., Tripakis, S. (eds.) FORMATS 2011. LNCS, vol. 6919, pp. 97–112. Springer, Heidelberg (2011)
7. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: A model-checking tool for real-time systems. In: Vardi, M.Y. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)

8. Cassez, F., Henzinger, T.A., Raskin, J.-F.: A comparison of control problems for timed and hybrid systems. In: Tomlin, C.J., Greenstreet, M.R. (eds.) HSCC 2002. LNCS, vol. 2289, pp. 134–148. Springer, Heidelberg (2002)
9. Daws, C., Yovine, S.: Two examples of verification of multirate timed automata with kronos. In: RTSS 1995, pp. 66–75. IEEE Computer Society Press (1995)
10. De Wulf, M., Doyen, L., Markey, N., Raskin, J.-F.: Robust safety of timed automata. FMSD 33(1-3), 45–84 (2008)
11. De Wulf, M., Doyen, L., Raskin, J.-F.: Almost ASAP semantics: From timed models to timed implementations. Formal Aspects of Computing 17(3), 319–341 (2005)
12. Henzinger, T.A., Sifakis, J.: The Embedded Systems Design Challenge. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 1–15. Springer, Heidelberg (2006)
13. Kordy, P., Langerak, R., Polderman, J.W.: Re-verification of a lip synchronization protocol using robust reachability. In: FMA, pp. 49–62 (2009)
14. Puri, A.: Dynamical properties of timed automata. Discrete Event Dynamic Systems 10(1-2), 87–113 (2000)
15. Sankur, O., Bouyer, P., Markey, N.: Shrinking timed automata. In: FSTTCS 2011, Leibniz-Zentrum für Informatik. LIPIcs, vol. 13, pp. 90–102 (2011)
16. Tripakis, S., Yovine, S.: Analysis of timed systems using time-abstracting bisimulations. Form. Methods Syst. Des. 18(1), 25–68 (2001)

Author Index