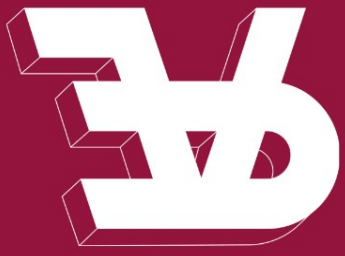
Tayssir Touili
Byron Cook
Paul Jackson (Eds.)

LNCS 6174

Computer Aided Verification

22nd International Conference, CAV 2010
Edinburgh, UK, July 2010
Proceedings



Springer

Lecture Notes in Computer Science 6174

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

Tayssir Touili Byron Cook Paul Jackson (Eds.)

Computer Aided Verification

22nd International Conference, CAV 2010
Edinburgh, UK, July 15-19, 2010
Proceedings

Springer

Volume Editors

Tayssir Touili
LIAFA, CNRS and University Paris Diderot
Case 7014, 75205 Paris Cedex 13, France
E-mail: touili@liafa.jussieu.fr

Byron Cook
Microsoft Research, Roger Needham Building
JJ Thomson Avenue, Cambridge CB3 0FB, UK
E-mail: bycook@microsoft.com

Paul Jackson
University of Edinburgh, School of Informatics
Edinburgh EH8 9AB, UK
E-mail: pbj@inf.ed.ac.uk

Library of Congress Control Number: 2010929755

CR Subject Classification (1998): F.3, D.2, D.3, D.2.4, F.4.1, C.2

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

ISSN 0302-9743
ISBN-10 3-642-14294-X Springer Berlin Heidelberg New York
ISBN-13 978-3-642-14294-9 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper 06/3180

Preface

This volume contains the proceedings of the 22nd International Conference on Computer-Aided Verification (CAV) held in Edinburgh, UK, July 15–19 2010. CAV is dedicated to the advancement of the theory and practice of computer-assisted formal analysis methods for software and hardware systems. The conference covers the spectrum from theoretical results to concrete applications, with an emphasis on practical verification tools and the algorithms and techniques that are needed for their implementation.

We received 145 submissions: 101 submissions of regular papers and 44 submissions of tool papers. These submissions went through a meticulous review process; each submission was reviewed by at least 4, and on average 4.2 Program Committee members. Authors had the opportunity to respond to the initial reviews during an author response period. This helped the Program Committee members to select 51 papers: 34 regular papers and 17 tool papers.

In addition to the accepted papers, the program also included:

- Five invited talks:
  - *Policy Monitoring in First-Order Temporal Logic,* by David Basin (ETH Zurich)
  - *Retrofitting Legacy Code for Security,* by Somesh Jha (University of Wisconsin-Madison)
  - *Induction, Invariants, and Abstraction,* by Deepak Kapur (University of New Mexico)
  - *Quantitative Information Flow: From Theory to Practice?* by Pasquale Malacaria (Queen Mary University) and
  - *Memory Management in Concurrent Algorithms,* by Maged Michael (IBM)
- Four invited tutorials:
  - *ABC: An Academic Industrial-Strength Verification Tool,* by Robert Brayton (University of California, Berkeley)
  - *Software Model Checking,* by Kenneth McMillan (Cadence Berkeley Labs)
  - *There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code,* by Thomas Reps (University of Wisconsin-Madison) and
  - *Constraint Solving for Program Verification: Theory and Practice by Example,* by Andrey Rybalchenko (Technische Universität München)

The program also included a session dedicated to the memory of Amir Pnueli, who died on November 2, 2009. Amir was one of the main leaders of modern advances in formal verification, and up to this year, he served on the CAV Steering Committee. His death is a big loss to our community. We dedicate these proceedings to his memory.

CAV 2010 was part of the Federated Logic Conference (FLoC 2010), hosted by the School of Informatics at the University of Edinburgh, Scotland. It was jointly organized with ICLP (International Conference on Logic Programming), IJCAR (International Joint Conference on Automated Reasoning), LICS (Logic in Computer Science), RTA (Rewriting Techniques and Applications), SAT (Theory and Applications of Satisfiability Testing), CSF (The Computer Security Foundations Symposium), and ITP (International Conference on Interactive Theorem Proving). In particular, the invited talks by David Basin and Deepak Kapur were, respectively, FLoC plenary and keynote talks.

CAV 2010 had eight affiliated workshops:

- The Fifth Automated Formal Methods Workshop (AFM 2010)
- Exploiting Concurrency Efficiently and Correctly (EC2-2010)
- Workshop on Evaluation Methods for Solvers, and Quality Metrics for Solutions (EMSQMS 2010)
- The First Hardware Verification Workshop (HWVW 2010)
- The Third International Workshop on Numerical Software Verification (NSV-3)
- The Second International Workshop on Practical Synthesis for Concurrent Systems (PSY 2010)
- International Workshop on Satisfiability Modulo Theories (SMT 2010)
- Synthesis, Verification and Analysis of Rich Models (SVARM 2010)

During the organization of CAV 2010, Edmund Clarke retired from the Steering Committee, and Orna Grumberg and Kenneth McMillan joined. Edmund Clarke was one of the founders of CAV, and we would like to especially thank him for his support of CAV from the start. We also thank the other Steering Committee members and the Chairs of CAV 2008 and CAV 2009 for their help and advice. We wish also to thank the Program Committee members and the external reviewers for their work in evaluating the submissions and assuring a high-quality program. We also thank Tomás Vojnar for his help in organizing the workshops. Finally, we thank Andrei Voronkov for creating and supporting the EasyChair conference management system.

CAV 2010 was supported by generous sponsorships. We gratefully acknowledge the support from Jasper Design Automation, IBM Research, Microsoft Research, NEC, EPSRC, NSF, Association for Symbolic Logic, CADE Inc., Google, Hewlett-Packard, and Intel.

July 2010

Tayssir Touili
Byron Cook
Paul Jackson

Conference Organization

Program Chairs

Tayssir Touili	LIAFA-CNRS, France
Byron Cook	Microsoft Research, UK
Paul Jackson	University of Edinburgh, UK

Program Committee

Rajeev Alur	University of Pennsylvania, USA
Domagoj Babić	UC Berkeley, USA
Christel Baier	Technical University of Dresden, Germany
Roderick Bloem	Graz University of Technology, Austria
Ahmed Bouajjani	LIAFA-University of Paris Diderot, France
Alessandro Cimatti	FBK-irst, Italy
Javier Esparza	Technische Universität München, Germany
Azadeh Farzan	University of Toronto, Canada
Martin Fränzle	University of Oldenburg, Germany
Ganesh Gopalakrishnan	University of Utah, USA
Mike Gordon	University of Cambridge, UK
Orna Grumberg	Technion, Israel
Ziyad Hanna	Jasper, USA
Holger Hermanns	Saarland University, Germany
Alan Hu	University of British Columbia, Canada
Kevin Jones	City University London, UK
Vineet Kahlon	NEC Labs, USA
Jean Krivine	PPS-CNRS, France
Daniel Kroening	Oxford University, UK
Sava Krstić	Intel Corporation, USA
Marta Kwiatkowska	Oxford University, UK
Oded Maler	VERIMAG-CNRS, France
Kenneth McMillan	Cadence, USA
David Monniaux	VERIMAG-CNRS, France
Markus Müller-Olm	Münster University, Germany
Kedar Namjoshi	Bell Labs, USA
Doron Peled	Bar Ilan University, Israel
Shaz Qadeer	Microsoft Research, USA
Jean-François Raskin	Brussels University, Belgium
Natasha Sharygina	University of Lugano, Switzerland

Helmut Veith	Vienna University of Technology, Austria
Kwangkeun Yi	Seoul National University, Korea
Karen Yorav	IBM Haifa, Israel
Greta Yorsh	IBM, USA

Steering Committee

Edmund M. Clarke	Carnegie Mellon University, USA
Mike Gordon	University of Cambridge, UK
Orna Grumberg	Technion, Israel
Robert P. Kurshan	Cadence, USA
Kenneth McMillan	Cadence, USA

Sponsors

Jasper Design Automation, IBM Research, Microsoft Research, NEC, EPSRC, NSF, Association for Symbolic Logic, CADE Inc., Google, Hewlett-Packard, Intel.

External Reviewers

Erika Ábrahám
Eli Arbel
Mohamed-Faouzi Atig
Jason Baumgartner
Jesse Bingham
Nikolaj Bjørner
Magnus Bjork
Sandrine Blazy
Pieter-Tjerk de Boer
Borzoo Bonakdarpour
Dragan Bosnacki
Matko Botincan
Patricia Bouyer-Decitre
Marco Bozzano
Tomas Brazdil
Angelo Brillout
Roberto Bruttomesso
Sebastian Burckhardt
Mike Case
Franck Cassez
Pavol Cerny
Hana Chockler
Frank Ciesinski
Ariel Cohen
Thomas Colcombet
Sylvain Conchon
Christopher Conway
Scott Cotton
Thao Dang
Vincent Danos
Aldric Degorre
Flavio M. De Paula
Henning Dierks
Antonio Dimalanta
Kyung-Goo Doh
Alastair Donaldson
Zhao Dong
Alexandre Donzé
Laurent Doyen
Klaus Dräger
Vijay D'Silva
Jeremy Dubreil
Andreas Eggers
Tayfun Elmas
Michael Emmi
Constantin Enea
Germain Faure
John Fearnley
Jérôme Feret
Alessandro Ferrante

Emmanuel Filiot
Seth Fogarthy
Anders Franzen
Goran Frehse
Laurent Fribourg
Vashti Galpin
Gilles Geeraerts
Steven German
Naghmeh Ghafari
Amit Goel
Alexey Gotsman
Susanne Graf
Karin Greimel
Andreas Griesmayer
Alberto Griggio
Marcus Groesser
Jim Grundy
Ashutosh Gupta
Dan Gutfreund
Peter Habermehl
Ernst Moritz Hahn
Leopold Haller
Philipp Haller
John Harrison
Arnd Hartmanns
John Hatcliff
Nannan He
Christian Herde
Hakan Hjort
Georg Hofferek
Andreas Holzer
William Hung
Hardi Hungar
Radu Iosif
Franjo Ivancić
Alexander Ivrii
Shahid Jabbar
Visar Januzaj
Bertrand Jeannet
Naiyong Jin
Barbara Jobstmann
Carson Jones
Yungbum Jung
Alexander Kaiser
Joost-Pieter Katoen
Mark Kattenbelt
Gal Katz
Jean-françois Kempf
Christian Kern
Sunghun Kim
Zachary Kincaid
Johannes Kinder
Joachim Klein
Sascha Klüppelholz
William Knottenbelt
Robert Könighofer
Soonho Kong
Maciej Koutny
Victor Kravets
Stephane Lafortune
Mario Lamberger
Peter Lammich
Cosimo Laneve
Frédéric Lang
François Laroussinie
Salvatore La Torre
Doug Lea
Oukseh Lee
Wonchan Lee
Axel Legay
Colas Le Guernic
Martin Leucker
Guodong Li
Henrik Lipskoch
Laurie Lugrin
Lars Lundgren
Parthasarathy Madhusudan
Rupak Majumdar
Nicolas Maquet
Johan Martensson
Radu Mateescu
Stephen McCamant
Bill McCloskey
Annabelle McIver
Igor Melatti
Yael Meller
Eric Mercer
Roland Meyer
Alan Mishchenko
John Moondanos

Leonardo de Moura
Sergio Mover
Matthieu Moy
Iman Narasamdya
Ziv Nevo
Dejan Nickovic
Gethin Norman
Hakjoo Oh
Avigail Orni
Rotem Oshman
Joel Ouaknine
Sungwoo Park
David Parker
Edgar Pek
Ruzica Piskac
Nir Piterman
Andreas Podelski
Corneliu Popeea
Mitra Purandare
Kairong Qian
Zvonimir Rakamarić
Yusi Ramadian
Stefan Ratschan
Jakob Rehof
Noam Rinetzky
Oleg Rokhlenko
Simone Rollini
Sitvanit Ruah
Philipp Rümmer
Marco Roveri
Andrey Rybalchenko
Sukyoung Ryu
Yaniv Sa'ar
Marko Samer
Sriram Sankaranarayanan
Gerald Sauter
Prateek Saxena
Christian Schallhart
Sven Schewe
Sylvain Schmitz
Viktor Schuppan
Stefan Schwoon
Alexander Serebrenik
Frédéric Servais
Ali Sezgin
Ohad Shacham
Subodh Sharma
Sarai Sheinvald
Mihaela Sighireanu
Nishant Sinha
Sebastian Skalberg
Jeremy Sproston
Stefan Staber
Mani Swaminathan
Nathalie Sznajder
Greg Szubzda
Paulo Tabuada
Murali Talupur
Michael Tautschnig
Tino Teige
PS Thiagarajan
Cesare Tinelli
Ashish Tiwari
Stefano Tonetta
Stavros Tripakis
Aliaksei Tsitovich
Frits Vaandrager
Viktor Vafeiadis
Martin Vechev
Tatyana Veksler
Mahesh Viswanathan
Yakir Vizel
Anh Vo
Björn Wachter
Thomas Wahl
Bow-Yaw Wang
Chao Wang
Andrzej Wasowski
Georg Weissenbacher
Alexander Wenner
Stephan Wilhelm
Ralf Wimmer
Christoph Wintersteiger
Verena Wolf
Nicolás Wolovick
Avi Yadgar
Eran Yahav
Sergio Yovine
Lijun Zhang
Florian Zuleger

Table of Contents

Invited Talks

Invited Tutorials

Session 1. Software Model Checking

Session 2. Model Checking and Automata

Session 3. Tools

Session 4. Counter and Hybrid Systems Verification

Session 5. Memory Consistency

Session 6. Verification of Hardware and Low Level Code

Session 7. Tools

Session 8. Synthesis

Session 9. Concurrent Program Verification I

Session 10. Compositional Reasoning

Session 11. Tools

Session 12. Decision Procedures

Session 13. Concurrent Program Verification II

Session 14. Tools

Policy Monitoring in First-Order Temporal Logic*

David Basin, Felix Klaedtke, and Samuel Müller

Department of Computer Science, ETH Zurich, Switzerland

Abstract. We present an approach to monitoring system policies. As a specification language, we use an expressive fragment of a temporal logic, which can be effectively monitored. We report on case studies in security and compliance monitoring and use these to show the adequacy of our specification language for naturally expressing complex, realistic policies and the practical feasibility of monitoring these policies using our monitoring algorithm.

1 Introduction

Runtime monitoring is an approach to verifying system properties at execution time by using an online algorithm to check whether a system trace satisfies a temporal property. Whereas novel application areas such as compliance or business activity monitoring [8,19,24] require expressive property specification languages, current monitoring techniques are restricted in the properties they can handle. They either support properties expressed in propositional temporal logics and thus cannot cope with variables ranging over infinite domains [11, 27,34,39,49], do not provide both universal and existential quantification [6,30, 40,43] or only in restricted ways [6,25,47,48], do not allow arbitrary quantifier alternation [6,38], cannot handle unrestricted negation [13,38,46], do not provide quantitative temporal operators [38,43], or cannot simultaneously handle both past and future operators [13,25,38–40,44,46,48].

In this paper, we present our recent work [9,10] on runtime monitoring using an expressive safety fragment of metric first-order temporal logic (MFOTL), which overcomes most of the above limitations. The fragment consists of formulae of the form $\Box\,\phi$, where ϕ is bounded, i.e., its temporal operators refer only finitely into the future. As both (metric) past and bounded future operators may be arbitrarily nested, MFOTL supports natural specifications of complex policies. Moreover, the monitors work with infinite structures where relations are either representable by automata, so-called automatic structures [12,32], or are finite.

We review MFOTL and our monitoring algorithm, present applications, and give performance results. For reasons of space, we only consider here monitoring structures with finite relations. In [10], we also show how to handle automatic structures and provide all definitions, algorithms, and proofs. Further details on our case studies and performance results are given in [9].

* This work was partially supported by the Nokia Research Center, Switzerland.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 1–18, 2010.

The applications we present come from the domain of security and compliance monitoring. An example, from financial reporting, is the requirement: *Every transaction of a customer who has within the last 30 days been involved in a previous suspicious transaction, must be reported as suspicious within two days.* Our examples illustrate MFOTL's suitability for specifying complex, realistic security policies. The class of policies covered constitute safety properties, where compliance can be checked by monitoring system traces. In the domain of security, this encompasses most traditional access-control policies as well as usage-control policies and policies arising in regulatory compliance. As we will see, such policies often combine event and state predicates, relations on data, and complex temporal relationships; all of these aspects can be naturally represented by MFOTL formulae interpreted over a metric, point-based semantics.

To evaluate our monitoring algorithm, we monitored different policies on synthetic data streams. Our experiments indicate that our approach is practically feasible with modest computing and storage requirements. Indeed, given that events can be processed in the order of milliseconds, the efficiency is such that our monitors can also be used online to detect policy violations.

2 Monitoring Metric First-Order Temporal Properties

We first introduce metric first-order temporal logic (MFOTL), an extension of propositional metric temporal logic [33]. Afterwards, we describe our monitoring algorithm from [10] for a safety fragment of MFOTL.

2.1 Metric Temporal First-Order Logic

Syntax and Semantics. Let $\mathbb{I}$ be the set of nonempty intervals over $\mathbb{N}$. We often write an interval in $\mathbb{I}$ as $[b, b') := \{a \in \mathbb{N} \mid b \leq a < b'\}$, where $b \in \mathbb{N}$, $b' \in \mathbb{N} \cup \{\infty\}$, and $b < b'$. A *signature* S is a tuple (C, R, ι), where C is a finite set of constant symbols, R is a finite set of predicates disjoint from C, and the function $\iota : R \to \mathbb{N}$ associates each predicate $r \in R$ with an arity $\iota(r) \in \mathbb{N}$. In the following, let $S = (C, R, \iota)$ be a signature and V a countably infinite set of variables, assuming $V \cap (C \cup R) = \emptyset$.

The *(MFOTL) formulae* over the signature S are given by the grammar

$$\phi ::= t_1 \approx t_2 \mid t_1 \prec t_2 \mid r(t_1, \ldots, t_{\iota(r)}) \mid \neg\phi \mid \phi \wedge \phi \mid \exists x.\phi \mid \bullet_I \phi \mid \bigcirc_I \phi \mid \phi \,\mathsf{S}_I\, \phi \mid \phi \,\mathsf{U}_I\, \phi\,,$$

where $t_1, t_2, \ldots$ range over the elements in $V \cup C$, and r, x, and I range over the elements in R, V, and $\mathbb{I}$, respectively.

To define MFOTL's semantics, we need the following notions. A *(first-order) structure* $\mathcal{D}$ over S consists of a domain $|\mathcal{D}| \neq \emptyset$ and interpretations $c^{\mathcal{D}} \in |\mathcal{D}|$ and $r^{\mathcal{D}} \subseteq |\mathcal{D}|^{\iota(r)}$, for each $c \in C$ and $r \in R$. A *temporal (first-order) structure* over S is a pair $(\bar{\mathcal{D}}, \bar{\tau})$, where $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \ldots)$ is a sequence of structures over S and $\bar{\tau} = (\tau_0, \tau_1, \ldots)$ is a sequence of natural numbers (i.e., time stamps), where:

1. The sequence $\bar{\tau}$ is monotonically increasing (i.e., $\tau_i \leq \tau_{i+1}$, for all $i \geq 0$) and makes progress (i.e., for every $i \geq 0$, there is some $j > i$ such that $\tau_j > \tau_i$).

$$
\begin{array}{lll}
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models t \approx t' & \text{iff} & v(t) = v(t') \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models t \prec t' & \text{iff} & v(t) < v(t') \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models r(t_1, \ldots, t_{\iota(r)}) & \text{iff} & \big(v(t_1), \ldots, v(t_{\iota(r)})\big) \in r^{\mathcal{D}_i} \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \neg\phi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \not\models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \phi \wedge \psi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \phi \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \psi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \exists x.\, \phi & \text{iff} & (\bar{\mathcal{D}}, \bar{\tau}, v[x/d], i) \models \phi, \text{ for some } d \in |\bar{\mathcal{D}}| \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \bullet_I \phi & \text{iff} & i > 0,\ \tau_i - \tau_{i-1} \in I, \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, v, i-1) \models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \bigcirc_I \phi & \text{iff} & \tau_{i+1} - \tau_i \in I \text{ and } (\bar{\mathcal{D}}, \bar{\tau}, v, i+1) \models \phi \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \phi\, \mathsf{S}_I\, \psi & \text{iff} & \text{for some } j \le i,\ \tau_i - \tau_j \in I,\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi, \\
 & & \text{and } (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \phi, \text{ for all } k \in [j+1, i+1) \\
(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \phi\, \mathsf{U}_I\, \psi & \text{iff} & \text{for some } j \ge i,\ \tau_j - \tau_i \in I,\ (\bar{\mathcal{D}}, \bar{\tau}, v, j) \models \psi, \\
 & & \text{and } (\bar{\mathcal{D}}, \bar{\tau}, v, k) \models \phi, \text{ for all } k \in [i, j)
\end{array}
$$

Fig. 1. Semantics of MFOTL

2. $\bar{\mathcal{D}}$ has constant domains, i.e., $|\mathcal{D}_i| = |\mathcal{D}_{i+1}|$, for all $i \ge 0$. We denote the domain by $|\bar{\mathcal{D}}|$ and require that $|\bar{\mathcal{D}}|$ is strict linearly ordered by a relation $<$.
3. Each constant symbol $c \in C$ has a rigid interpretation, i.e., $c^{\mathcal{D}_i} = c^{\mathcal{D}_{i+1}}$, for all $i \ge 0$. We denote c's interpretation by $c^{\bar{\mathcal{D}}}$.

A *valuation* is a mapping $v : V \to |\bar{\mathcal{D}}|$. We abuse notation by applying a valuation v also to constant symbols $c \in C$, with $v(c) = c^{\bar{\mathcal{D}}}$. For a valuation v, the variable vector $\bar{x} = (x_1, \ldots, x_n)$, and $\bar{d} = (d_1, \ldots, d_n) \in |\bar{\mathcal{D}}|^n$, $v[\bar{x}/\bar{d}]$ is the valuation mapping x_i to d_i, for $1 \le i \le n$, and the other variables' valuation is unaltered.

The semantics of MFOTL, $(\bar{\mathcal{D}}, \bar{\tau}, v, i) \models \phi$, is given in Figure 1, where $(\bar{\mathcal{D}}, \bar{\tau})$ is a temporal structure over the signature S, with $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \ldots)$, $\bar{\tau} = (\tau_0, \tau_1, \ldots)$, v a valuation, $i \in \mathbb{N}$, and ϕ a formula over S. Note that the temporal operators are augmented with intervals and a formula of the form $\bullet_I \phi$, $\bigcirc_I \phi$, $\phi \mathsf{S}_I \psi$, or $\phi \mathsf{U}_I \psi$ is only satisfied in $(\bar{\mathcal{D}}, \bar{\tau})$ at the time point i if it is satisfied within the bounds given by the interval I of the respective temporal operator, which are relative to the current time stamp τ_i.

Terminology and Notation. We use standard syntactic sugar such as $\blacksquare_I \phi := \neg(\mathit{true}\, \mathsf{S}_I\, \neg\phi)$ and $\square_I \phi := \neg(\mathit{true}\, \mathsf{U}_I\, \neg\phi)$, where $\mathit{true} := \exists x.\, x \approx x$. We also use non-metric operators like $\square \phi := \square_{[0,\infty)} \phi$. We omit parentheses where possible, e.g., unary operators (temporal and Boolean) bind stronger than binary ones.

We call formulae with no temporal operators *first-order*. A formula α is *bounded* if the interval I of every temporal operator U_I occurring in α is finite. The outermost connective (i.e., Boolean connective, quantifier, or temporal operator) occurring in a formula α is called the *main connective* of α. A formula that has a temporal operator as its main connective is a *temporal* formula. The set $\mathit{tsub}(\alpha)$ of *immediate* temporal subformulae of α is: (i) $\mathit{tsub}(\beta)$, if $\alpha = \neg\beta$ or $\alpha = \exists x.\, \beta$, (ii) $\mathit{tsub}(\beta) \cup \mathit{tsub}(\gamma)$, if $\alpha = \beta \wedge \gamma$, (iii) $\{\alpha\}$, if α is a temporal formula, and (iv) $\emptyset$ otherwise. For instance, for $\alpha := ((\bigcirc \beta)\, \mathsf{S}\, \gamma) \wedge \bullet \beta'$, we have $\mathit{tsub}(\alpha) = \mathit{tsub}((\bigcirc \beta)\, \mathsf{S}\, \gamma) \cup \mathit{tsub}(\bullet \beta') = \{(\bigcirc \beta)\, \mathsf{S}\, \gamma, \bullet \beta'\}$.

For a formula α with the free variables $\bar{x} = (x_1, \ldots, x_n)$, we define the set of satisfying elements at time point $i \in \mathbb{N}$ in the temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$ as

$$\alpha^{(\bar{\mathcal{D}},\bar{\tau},i)} := \left\{ \bar{d} \in |\bar{\mathcal{D}}|^n \;\middle|\; (\bar{\mathcal{D}}, \bar{\tau}, v[\bar{x}/\bar{d}], i) \models \alpha, \text{ for some valuation } v \right\}.$$

If α is first-order, $\alpha^{(\bar{\mathcal{D}},\bar{\tau},i)}$ only depends on the structure $\mathcal{D}_i$ and we just write $\alpha^{\mathcal{D}_i}$ in this case.

2.2 Monitoring

In the following, let Ψ be an MFOTL formula over the signature $S = (C, R, \iota)$. To effectively monitor Ψ, we restrict both the formula Ψ and the temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$ over S, where $\bar{\mathcal{D}} = (\mathcal{D}_0, \mathcal{D}_1, \ldots)$ and $\bar{\tau} = (\tau_0, \tau_1, \ldots)$. To begin with, we require Ψ to be of the form $\Box \Psi'$, where Ψ' is bounded.[1] To detect violations, prior to monitoring, we try to rewrite $\neg\Psi'$ to a logically equivalent formula Φ, belonging to a syntactically-defined fragment. The monitoring algorithm then iteratively processes the temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$, evaluating Φ at each time point. Note that to identify violations, Ψ usually contains free variables and the violations are the satisfying assignments of Φ, which the monitor outputs.

The reason for rewriting $\neg\Psi'$ to Φ, rather than using $\neg\Psi'$ directly, is that the monitoring algorithm stores intermediate results when processing $(\bar{\mathcal{D}}, \bar{\tau})$ and therefore these results must be finite relations.[2] In particular, every relation $r^{\mathcal{D}_i}$ must be finite, for $i \in \mathbb{N}$ and $r \in R$. With the restriction to finite relations, we inherit a standard problem from database theory [3]. Namely, when $|\bar{\mathcal{D}}|$ is infinite, a query with negation can have an infinite answer set that itself cannot be represented by a finite relation. The restriction to so-called domain-independent queries, i.e., queries for which the answer set only depends on elements that occur in the database, only partially solves the problem: This guarantees finiteness but checking domain independence is undecidable [22]. A standard approach taken in database theory is therefore to try to rewrite a query into a form that falls into a syntactically-defined fragment that guarantees both the domain independence and the finiteness of the intermediate results. We take this approach and further details on rewrite rules and such a syntactically-defined fragment for MFOTL can be found in [10]. In the remainder of this section, we assume that Φ is from this monitorable fragment.

Overview. Our monitoring algorithm incrementally builds a sequence of structures $\hat{\mathcal{D}}_0, \hat{\mathcal{D}}_1, \ldots$ over an extended signature $\hat{S}$. The extension depends on the

[1] It follows that Ψ describes a safety property. Note, however, there are safety properties expressible in MFOTL that do not have such a syntactic form [15]. This is in contrast to propositional linear temporal logic, where every ω-regular safety property can be expressed as a formula $\Box\beta$, where β contains only past operators [36].

[2] In fact, a weaker requirement suffices, namely, each $\mathcal{D}_i$ is an automatic structure [12, 32] and the $\mathcal{D}_i$s are uniformly represented. When using automatic structures, no further requirements on Ψ' are necessary and our monitoring algorithm can work with any Φ that is logically equivalent to $\neg\Psi'$. The intermediate results are also "automatic" and effectively computable [10].

temporal subformulae of Φ. For each time point i, we determine the elements that satisfy Φ by evaluating a first-order formula $\hat{\Phi}$ over $\hat{\mathcal{D}}_i$. Observe that for a temporal subformula with a future operator as its main connective, we usually cannot yet carry out this evaluation at time point i. The monitoring algorithm therefore maintains a queue of unevaluated formulae and evaluates them when enough time has elapsed.

We describe first how we extend S and transform Φ. Afterwards, we explain how we incrementally build $\hat{\mathcal{D}}_i$. Finally, we present our monitoring algorithm. For the ease of exposition, we assume in the following that the temporal subformulae of Φ are of the form $\beta \,\mathsf{S}_I\, \gamma$ and $\Box_{[0,b)}\, \beta$. The more general case for the temporal operator U_I is along the same lines as $\Box_{[0,b)}$ but is technically more involved. The cases for $\bullet_I$ and $\bigcirc_I$ are straightforward and omitted here.

Signature Extension and Structure Construction. The extended signature $\hat{S}$ contains all constants and predicates in S, with the same arities. Moreover, for each temporal subformula α of Φ, $\hat{S}$ includes the new auxiliary predicates p_α and r_α, of arities n and $n+1$ respectively, where n is the number of free variables in α. For θ, a subformula of Φ over the signature S, $\hat{\theta}$ denotes the transformed formula over $\hat{S}$, where each $\alpha \in \mathit{tsub}(\theta)$ with the free variables $\bar{x}$ is replaced by $p_\alpha(\bar{x})$.

For $i \in \mathbb{N}$, $c \in C$, and $r \in R$, we define $|\hat{\mathcal{D}}_i| := |\bar{\mathcal{D}}| \cup \mathbb{N}$, $c^{\hat{\mathcal{D}}_i} := c^{\mathcal{D}_i}$, and $r^{\hat{\mathcal{D}}_i} := r^{\mathcal{D}_i}$. The auxiliary relations in the $\hat{\mathcal{D}}_i$s are defined inductively over both time and the formula structure. Furthermore, their construction is incremental in the sense that it reuses the auxiliary relations from the previous time points.

We start with the auxiliary relations for a subformula α of the form $\beta \,\mathsf{S}_{[b,b')}\, \gamma$. The non-metric variant of the construction reflects that $\beta \,\mathsf{S}\, \gamma$ is logically equivalent to $\gamma \vee \beta \wedge \bullet(\beta \,\mathsf{S}\, \gamma)$: For $i \geq 0$ and assuming without loss of generality that β and γ have the same vector of free variables, we define

$$p_{\beta \mathsf{S} \gamma}^{\hat{\mathcal{D}}_i} := \hat{\gamma}^{\hat{\mathcal{D}}_i} \cup \begin{cases} \emptyset & \text{if } i = 0, \\ \hat{\beta}^{\hat{\mathcal{D}}_i} \cap p_{\beta \mathsf{S} \gamma}^{\hat{\mathcal{D}}_{i-1}} & \text{if } i > 0. \end{cases}$$

Observe that this definition only depends on the relations in $\hat{\mathcal{D}}_i$ for which the corresponding predicates occur in the subformulae of $\hat{\beta}$ or $\hat{\gamma}$, and on the auxiliary relation $p_{\beta \mathsf{S} \gamma}^{\hat{\mathcal{D}}_{i-1}}$, when $i > 0$.

To incorporate the timing constraint for the interval $[b, b')$ of the S operator, we first incrementally construct the auxiliary relations for r_α similar to the definition above: For $i \geq 0$, we define $r_\alpha^{\hat{\mathcal{D}}_i} := N \cup U$, where $N := \hat{\gamma}^{\hat{\mathcal{D}}_i} \times \{0\}$ and

$$U := \begin{cases} \emptyset & \text{if } i = 0, \\ \left\{ (\bar{a}, y) \;\middle|\; \bar{a} \in \hat{\beta}^{\hat{\mathcal{D}}_i},\ y < b',\ \text{and } (\bar{a}, y + \tau_{i-1} - \tau_i) \in r_\alpha^{\hat{\mathcal{D}}_{i-1}} \right\} & \text{if } i > 0. \end{cases}$$

Intuitively, a pair $(\bar{a}, y)$ is in $r_\alpha^{\hat{\mathcal{D}}_i}$ if $\bar{a}$ satisfies α at the time point i independent of the lower bound b, where the "age" y indicates how long ago the formula α was satisfied by $\bar{a}$. If $\bar{a}$ satisfies γ at the time point i, it is added to $r_\alpha^{\hat{\mathcal{D}}_i}$ with the age 0. For $i > 0$, we also update the tuples $(\bar{a}, y) \in r_\alpha^{\hat{\mathcal{D}}_{i-1}}$ when $\bar{a}$ satisfies β at time point i, i.e., the age is adjusted by the difference of the time stamps τ_{i-1}

and τ_i in case the new age is less than b'. Otherwise it is too old to satisfy α and the updated tuple is not included in $r_\alpha^{\hat{\mathcal{D}}_i}$.

Finally, we obtain the auxiliary relation $p_\alpha^{\hat{\mathcal{D}}_i}$ from $r_\alpha^{\hat{\mathcal{D}}_i}$ by checking whether the age of a tuple in $r_\alpha^{\hat{\mathcal{D}}_i}$ is old enough:

$$p_\alpha^{\hat{\mathcal{D}}_i} := \left\{ \bar{a} \;\middle|\; (\bar{a}, y) \in r_\alpha^{\hat{\mathcal{D}}_i}, \text{ for some } y \geq b \right\}.$$

We now address the bounded future operator $\Box_{[0,b)}$, with $b \in \mathbb{N} \setminus \{0\}$. Assume that $\alpha = \Box_{[0,b)}\, \beta$. For $i \in \mathbb{N}$, let $\ell_i := \max\{k \in \mathbb{N} \mid \tau_{i+k} - \tau_i < b\}$ denote the *lookahead offset* at time point i. Note that only $\hat{\beta}^{\hat{\mathcal{D}}_i}, \ldots, \hat{\beta}^{\hat{\mathcal{D}}_{i+\ell_i}}$ are relevant for determining $\alpha^{(\bar{\mathcal{D}},\bar{\tau},i)}$. For $i \in \mathbb{N}$, we could directly define $p_\alpha^{\hat{\mathcal{D}}_i}$ as $\bigcap_{j\in\{0,\ldots,\ell_i\}} \hat{\beta}^{\hat{\mathcal{D}}_{i+j}}$. However, this construction has the drawback that for i and $i+1$, we must recompute the intersections of the $\hat{\beta}^{\hat{\mathcal{D}}_{i+j}}$s for $j \in \{1, \ldots, \ell_i\}$.

We instead define $p_\alpha^{\hat{\mathcal{D}}_i}$ in terms of the incrementally-built auxiliary relation $r_\alpha^{\hat{\mathcal{D}}_i}$, where $(\bar{a}, k) \in r_\alpha^{\hat{\mathcal{D}}_i}$ iff $\bar{a} \in \hat{\beta}^{\hat{\mathcal{D}}_{i+j}}$, for all $j \in \{k, \ldots, \ell_i\}$. As before, we construct $r_\alpha^{\hat{\mathcal{D}}_i}$ from two sets N and U. N contains the elements from the new time points $i + \ell_{i-1}, \ldots, i + \ell_i$, where $\ell_{-1} := 0$ for convenience. U contains the updated elements from $r_\alpha^{\hat{\mathcal{D}}_{i-1}}$, if $i > 0$. To update an element $(\bar{a}, k) \in r_\alpha^{\hat{\mathcal{D}}_{i-1}}$, we check that $\bar{a}$ also satisfies β at the new time points. Furthermore, we decrease its index k, if $k > 0$. Formally, for $i \geq 0$, we define $r_\alpha^{\hat{\mathcal{D}}_i} := N \cup U$, where

$$N := \left\{ (\bar{a}, k) \;\middle|\; \ell_{i-1} \leq k \leq \ell_i \text{ and } \bar{a} \in \hat{\beta}^{\hat{\mathcal{D}}_{i+k+j}}, \text{ for all } j \in \mathbb{N} \text{ with } k + j \leq \ell_i \right\}$$

and $U := \emptyset$ when $i = 0$ and if $i > 0$, then

$$U := \left\{ (\bar{a}, \max\{0, k-1\}) \;\middle|\; (\bar{a}, k) \in r_\alpha^{\hat{\mathcal{D}}_{i-1}} \text{ and if } \ell_i - \ell_{i-1} \geq 0 \text{ then } (\bar{a}, \ell_{i-1}) \in N \right\}.$$

Finally, we define $p_\alpha^{\hat{\mathcal{D}}_i} := \{\bar{a} \mid (\bar{a}, 0) \in r_\alpha^{\hat{\mathcal{D}}_i}\}$.

We remark that both constructions of the auxiliary relations for the subformulae for the forms $\beta\, \mathsf{S}_I\, \gamma$ and $\Box_{[0,b)}\, \beta$ can be optimized. For example, we can delete a tuple $(\bar{a}, k)$ in $r_{\Box_{[0,b)}\,\beta}^{\hat{\mathcal{D}}_i}$ if it also contains a tuple $(\bar{a}, k')$ with $k' < k$.

Example. Before presenting our monitoring algorithm, we illustrate the formula transformation and the constructions of the auxiliary relations with the formula

$$\Box \forall x.\, in(x) \rightarrow \Diamond_{[0,6)}\, out(x)\,.$$

To detect violations, we negate this formula and push negation inwards. To determine which elements violate the property, we also drop the quantifier, obtaining the formula $\Diamond \left(in(x) \wedge \Box_{[0,6)}\, \neg out(x) \right)$. Since relations for *out* are finite, $\neg out(x)$ describes an infinite set and therefore the auxiliary relations for the subformula $\Box_{[0,6)}\, \neg out(x)$ are infinite. Hence, we further rewrite the formula into the logically equivalent formula $\Diamond \Phi$, with $\Phi := in(x) \wedge \Box_{[0,6)} \left(\neg out(x) \wedge \blacklozenge_{[0,6)}\, in(x) \right)$. The formula Φ is in our monitorable MFOTL fragment.

Observe that $\alpha := \Box_{[0,6)} \left(\neg out(x) \wedge \blacklozenge_{[0,6)}\, in(x) \right)$ and $\alpha' := \blacklozenge_{[0,6)}\, in(x)$ are the only temporal subformulae of Φ. The transformed formula $\hat{\Phi} = in(x) \wedge p_\alpha(x)$

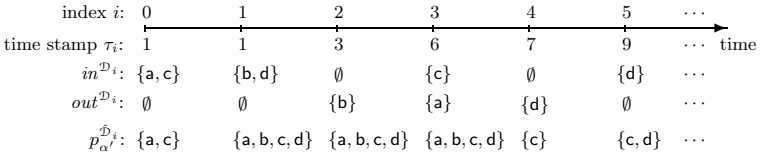


Fig. 2. A temporal structure

is over the signature $\hat{S}$ that extends Φ's signature with the auxiliary unary predicates p_α and $p_{\alpha'}$ and the binary predicates r_α and $r_{\alpha'}$.

We only illustrate the incremental constructions of the auxiliary relations for α by considering the temporal structure in Figure 2, which also depicts the relations for $p_{\alpha'}$. Observe that to build the relations $r_\alpha^{\hat{\mathcal{D}}_i}$, for $i \geq 0$, we require the relations $out^{\mathcal{D}_{i+k}}$ and the auxiliary relations $p_{\alpha'}^{\hat{\mathcal{D}}_{i+k}}$ with $k \in \{0, \ldots, \ell_i\}$, for the lookahead offset ℓ_i at time point i.

For the time point $i = 0$, we have $\ell_0 = 3$ because $\tau_3 - \tau_0 < 6$ and $\tau_4 - \tau_0 = 6$. Furthermore, the auxiliary relation $r_\alpha^{\hat{\mathcal{D}}_0}$ is $\{(\mathsf{c}, k) | 0 \leq k \leq 3\} \cup \{(\mathsf{d}, k) | 1 \leq k \leq 3\}$. In general, a pair $(\bar{a}, k)$ is in $r_\alpha^{\hat{\mathcal{D}}_i}$ iff $\bar{a}$ did not occur in one of the relations $out^{\mathcal{D}_{i+j}}$, with $j \in \{k, \ldots, \ell_i\}$ and $\bar{a}$ previously appeared in $in^{\mathcal{D}_{j'}}$, for some $j' \leq i + j$ with $\tau_{i+j} - \tau_{j'} < 6$. For example, the pair $(\mathsf{c}, 2)$ is in $r_\alpha^{\hat{\mathcal{D}}_0}$, since c is not in $out^{\mathcal{D}_2} \cup out^{\mathcal{D}_3}$ and c is in $in^{\mathcal{D}_0}$ and hence in $p_{\alpha'}^{\hat{\mathcal{D}}_2}$ and $p_{\alpha'}^{\hat{\mathcal{D}}_3}$. Recall that the lookahead offset ℓ_0 is 3 and therefore we only look at the time points 0 through 3. We obtain $p_\alpha^{\hat{\mathcal{D}}_0}$ as $\{\bar{a} \mid (\bar{a}, 0) \in r_\alpha^{\hat{\mathcal{D}}_0}\} = \{\mathsf{c}\}$, which contains also the satisfying elements for Φ at time point 0, since $p_\alpha^{\hat{\mathcal{D}}_0} \cap in^{\mathcal{D}_0} = \{\mathsf{c}\}$.

For the time point $i = 1$, the lookahead offset ℓ_1 is 2. Since $\ell_1 = \ell_0 - 1$, we need not consider any new time points, i.e., we obtain $r_\alpha^{\hat{\mathcal{D}}_1}$ from $r_\alpha^{\hat{\mathcal{D}}_0}$ by updating the tuples contained in $r_\alpha^{\hat{\mathcal{D}}_0}$, yielding $r_\alpha^{\hat{\mathcal{D}}_1} = \{(\mathsf{c}, 0), (\mathsf{c}, 1), (\mathsf{c}, 2), (\mathsf{d}, 0), (\mathsf{d}, 1), (\mathsf{d}, 2)\}$ and $p_\alpha^{\hat{\mathcal{D}}_1} = \{\mathsf{c}, \mathsf{d}\}$. The corresponding set of violating elements is $p_\alpha^{\hat{\mathcal{D}}_1} \cap in^{\mathcal{D}_1} = \{\mathsf{d}\}$. For the time point $i = 2$, we must also account for the new time point 4, since $\ell_2 = 2$. The only new element in $r_\alpha^{\hat{\mathcal{D}}_2}$ is $(\mathsf{c}, 2)$. The updated elements are $(\mathsf{c}, 0)$ and $(\mathsf{c}, 1)$. The pairs in $r_\alpha^{\hat{\mathcal{D}}_1}$ with the first component d are not updated since $\mathsf{d} \in out^{\mathcal{D}_4}$. We therefore obtain $p_\alpha^{\hat{\mathcal{D}}_2} = \{\mathsf{c}\}$ and $p_\alpha^{\hat{\mathcal{D}}_2} \cap in^{\mathcal{D}_2} = \emptyset$.

The Monitoring Algorithm. Figure 3 presents our monitoring algorithm M_Φ. Without loss of generality, we assume that each temporal subformula occurs only once in Φ. In the following, we describe M_Φ's operation.

M_Φ uses two counters ℓ and i. The counter ℓ is the index of the current element $(\mathcal{D}_\ell, \tau_\ell)$ in the input sequence $(\mathcal{D}_0, \tau_0), (\mathcal{D}_1, \tau_1), \ldots$, which is processed sequentially. Initially, ℓ is 0 and it is incremented with each loop iteration (lines 4–14). The counter $i \leq \ell$ is the index of the next time point i (possibly in the past, from ℓ's point of view) for which we evaluate $\hat{\Phi}$ over the structure $\hat{\mathcal{D}}_i$. The evaluation is delayed until the relations $p_\alpha^{\hat{\mathcal{D}}_i}$ for $\alpha \in tsub(\Phi)$ have all been built (lines 10–12).

1 $\ell \leftarrow 0$ *% current index in input sequence* $(\mathcal{D}_0, \tau_0), (\mathcal{D}_1, \tau_1), \ldots$
2 $i \leftarrow 0$ *% index of next query evaluation in input sequence* $(\mathcal{D}_0, \tau_0), (\mathcal{D}_1, \tau_1), \ldots$
3 $Q \leftarrow \{(\alpha, 0, \mathit{waitfor}(\alpha)) \mid \alpha \text{ is a temporal subformula of } \Phi\}$
4 **loop**
5 Carry over constants and relations of $\mathcal{D}_\ell$ to $\hat{\mathcal{D}}_\ell$.
6 **forall** $(\alpha, j, \emptyset) \in Q$ **do** *% respect ordering of subformulae*
7 *Build auxiliary relations* $p_\alpha^{\hat{\mathcal{D}}_j}$ *and* $r_\alpha^{\hat{\mathcal{D}}_j}$.
8 *Discard auxiliary relations that were involved in the construction of* $r_\alpha^{\hat{\mathcal{D}}_j}$.
9 **while** *all auxiliary relations* $p_\alpha^{\hat{\mathcal{D}}_i}$ *are built for* $\alpha \in \mathit{tsub}(\Phi)$ **do** *% eval query*
10 *Output* $(\hat{\Phi})^{\hat{\mathcal{D}}_i}$ *and* τ_i.
11 *Discard structure* $\hat{\mathcal{D}}_{i-1}$, *if* $i > 0$.
12 $i \leftarrow i + 1$
13 $Q \leftarrow \{(\alpha, \ell + 1, \mathit{waitfor}(\alpha)) \mid \alpha \text{ is a temporal subformula of } \Phi\} \cup \{(\alpha, j, \bigcup_{\alpha' \in \mathit{update}(S, \tau_{\ell+1} - \tau_\ell)} \mathit{waitfor}(\alpha')) \mid (\alpha, j, S) \in Q \text{ and } S \neq \emptyset\}$
14 $\ell \leftarrow \ell + 1$ *% process next element* $(\mathcal{D}_{\ell+1}, \tau_{\ell+1})$ *in input sequence*

Fig. 3. Monitoring algorithm M_Φ

Furthermore, M_Φ uses the list[3] Q to ensure that the auxiliary relations of $\hat{\mathcal{D}}_0, \hat{\mathcal{D}}_1, \ldots$ are built at the right time: if $(\alpha, j, \emptyset)$ is an element of Q at the beginning of a loop iteration, enough time has elapsed to build the relations for the temporal subformula α of the structure $\hat{\mathcal{D}}_j$. M_Φ initializes Q in line 3. The function *waitfor* identifies the subformulae that delay the formula evaluation:

$$\mathit{waitfor}(\alpha) := \begin{cases} \mathit{waitfor}(\beta) & \text{if } \alpha = \neg\beta \text{ or } \alpha = \exists x.\, \beta, \\ \mathit{waitfor}(\beta) \cup \mathit{waitfor}(\gamma) & \text{if } \alpha = \beta \wedge \gamma \text{ or } \alpha = \beta\, \mathsf{S}_I\, \gamma, \\ \{\alpha\} & \text{if } \alpha = \Box_{[0,b)}\, \beta, \\ \emptyset & \text{otherwise.} \end{cases}$$

The list Q is updated in line 13 before we increment ℓ and start a new loop iteration. For an update, we use the set $\mathit{update}(U, t)$ defined as

$$\{\Box_{[0,b-t)}\, \beta \mid \Box_{[0,b)}\, \beta \in U \text{ and } b - t > 0\} \cup \{\beta \mid \Box_{[0,b)}\, \beta \in U \text{ and } b - t \leq 0\},$$

where U is a set of formulae and $t \in \mathbb{N}$. The update adds a new tuple $(\alpha, \ell + 1, \mathit{waitfor}(\alpha))$ to Q, for each temporal subformula α of Φ, and it removes the tuples of the form $(\alpha, j, \emptyset)$ from Q. Moreover, for tuples (α, j, S) with $S \neq \emptyset$, the set S is updated using the functions *waitfor* and *update*, accounting for the elapsed time to the next time point, i.e. $\tau_{\ell+1} - \tau_\ell$.

In lines 6–8, we build the relations for which enough time has elapsed, i.e., the auxiliary relations for α in $\hat{\mathcal{D}}_j$ with $(\alpha, j, \emptyset) \in Q$. Since a tuple $(\alpha', j, \emptyset)$ does not occur before a tuple $(\alpha, j, \emptyset)$ in Q, where α is a subformula of α', the

[3] We abuse notation by using set notation for lists. Moreover, we assume that Q is ordered so that (α, j, S) occurs before (α', j', S'), whenever α is a proper subformula of α', or $\alpha = \alpha'$ and $j < j'$.

relations in $\hat{\mathcal{D}}_j$ for α are built before those for α'. To build the relations, we use the incremental constructions described earlier in this section. After we have built these relations for α in $\hat{\mathcal{D}}_j$, we discard relations no longer needed to reduce space consumption. For instance, if $j > 0$ and $\alpha = \beta \,\mathsf{S}_I\, \gamma$, then we discard the relations $r_\alpha^{\hat{\mathcal{D}}_{j-1}}$ and $p_{\alpha'}^{\hat{\mathcal{D}}_j}$ with $\alpha' \in tsub(\beta) \cup tsub(\gamma)$.

In lines 9–12, if the auxiliary relations for p_α in $\hat{\mathcal{D}}_i$ of all immediate temporal subformulae α of Φ have been built, then M_Φ outputs the valuations violating Ψ' at time point i together with τ_i. Furthermore, after each output, the remainder of the extended structure $\hat{\mathcal{D}}_{i-1}$ is discarded (if $i > 0$) and i is incremented by 1.

Note that because M_Φ does not terminate, it is not an algorithm in the strict sense. However, it effectively computes the elements violating Ψ', for every time point n. More precisely, whenever M_Φ outputs the set $(\hat{\Phi})^{\hat{\mathcal{D}}_i}$ in line 10, then this set is finite, effectively computable, and $(\hat{\Phi})^{\hat{\mathcal{D}}_i} = (\neg\Psi')^{(\bar{\mathcal{D}},\bar{\tau},i)}$. Moreover, for each $n \in \mathbb{N}$, M_Φ eventually sets the counter i to n in some loop iteration.

Since M_Φ iteratively processes the structures and time stamps in the temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$, we measure its memory usage with respect to the processed prefix of $(\bar{\mathcal{D}}, \bar{\tau})$. The counters ℓ and i are at most the length of the processed prefix. Hence, in the nth loop iteration, we need $O(\log n)$ bits for these two counters. We can modify the monitoring algorithm M_Φ so that it is independent of the prefix length by replacing the two counters with a single counter that stores $\ell - i$, i.e., the distance of ℓ from i. Since the list Q stores tuples that contain indices of the processed prefix, we must make them relative to the next query evaluation. Under the additional assumption that there are at most m equal time stamps in $\bar{\tau}$, the number of bits for the new counter is then logarithmically bounded by the maximal lookahead offset, which is at most $m \cdot s$, where s is the sum of the upper bounds of the intervals of the future operators occurring in Φ. Furthermore, the number of elements in the list Q is bounded by $m \cdot s \cdot k$, where k is the number of temporal subformulae in Φ. Most importantly, the number of elements in the auxiliary relations that M_Φ stores in the nth loop iteration can be polynomially bounded by m, s, k, and the cardinality of the *active domain* of the processed prefix, where $adom_\ell(\bar{\mathcal{D}}) := \{c^{\bar{\mathcal{D}}} \mid c \in C\} \cup \bigcup_{0 \le n \le \ell} \bigcup_{r \in R} \{d_j \mid (d_1, \ldots, d_{\iota(r)}) \in r^{\mathcal{D}_n} \text{ and } 1 \le j \le \iota(r)\}$. The degree of the polynomial is linear in the maximal number of free variables occurring in a temporal subformula of Φ. To achieve this polynomial bound, we must optimize the incremental construction of the auxiliary relations for $r_{\beta \mathsf{S}_{[b,\infty)}\gamma}$ so that the age of an element is the minimum of its actual age and the interval's lower bound b.

Given the above modifications to M_Φ and the additional assumption on the number of equal time stamps, the monitor's memory usage is polynomially bounded and independent of the length of the processed prefix. Moreover, the bound on the cardinalities of the auxiliary relations is independent of how often an element of $|\bar{\mathcal{D}}|$ appears in the relations of the processed prefix of the given temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$.

3 Case Studies

We have carried out several case studies where we formalized and monitored a wide range of policies from the domain of security and regulatory compliance. In the following, we give two representative examples and report on the monitors' runtime performance for these cases. Other examples are given in [9].

3.1 Approval Requirements

Consider a policy governing the publication of business reports within a company, where all reports must be approved prior to publication. A simplified form of such a policy might be

$$\Box \forall f.\, \mathit{publish}(f) \rightarrow \blacklozenge\, \mathit{approve}(f)\,.$$

But this is too simplistic. More realistically, we would also require, for example, that the person publishing the report must be an accountant and the person approving the publication must be the accountant's manager. Moreover, the approval must happen within, say, 10 days prior to publication.

Note that predicates like approving a report and being someone's manager differ in the following respect. The act of approving a report represents an *event*: It happens at a time point and does not have a duration. In contrast, being someone's manager describes a *state* that has a duration. Since MFOTL's semantics is point-based, it naturally captures events. Entities like system states do not have a direct counterpart in MFOTL. However, we can model them using start and finish events. The following formalization of the above policy illustrates these two different kinds of entities and how we deal with them in MFOTL. To distinguish between them, we use the terms *event predicate* and *state predicate*.

Signature. The signature consists of the unary relation symbols acc_S and acc_F, and the binary relation symbols mgr_S, mgr_F, *publish*, and *approve*. Intuitively, $\mathit{mgr}_S(m, a)$ marks the time when m becomes a's manager and $\mathit{mgr}_F(m, a)$ marks the corresponding finishing time. Analogously, $\mathit{acc}_S(a)$ and $\mathit{acc}_F(a)$ mark the starting and finishing times of a being an accountant. We use these predicates to simulate state predicates in MFOTL, e.g., the formula $\underline{\mathit{acc}}(a) := \neg \mathit{acc}_F(a)\ \mathsf{S}\ \mathit{acc}_S(a)$ holds at the time points where a is an accountant. It states that a starting event for a being an accountant has previously occurred and the corresponding finishing event has not occurred since then. Analogously, $\underline{\mathit{mgr}}(m, a) := \neg \mathit{mgr}_F(m, a) \mathsf{S} \mathit{mgr}_S(m, a)$ is the state predicate expressing that m is a's manager.

Formalization. Before formalizing the approval policy, we formalize assumptions about the start and finish events in a temporal structure $(\bar{\mathcal{D}}, \bar{\tau})$. These assumptions reflect the system requirement that these events are generated in a "well-formed" way. First, we assume that start and finish events do not occur at the same time point, since their ordering would then be unclear. For example, for the start and finish events of being an accountant, we assume that $(\bar{\mathcal{D}}, \bar{\tau})$ satisfies the formula

$$\Box \forall a.\, \neg\big(\mathit{acc}_S(a) \wedge \mathit{acc}_F(a)\big)\,.$$

Furthermore, we assume that every finish event is preceded by a matching start event and between two start events there is a finish event. Formally, for the start and finish events of being an accountant, we assume that $(\bar{\mathcal{D}}, \bar{\tau})$ satisfies

$$\Box \forall a.\, acc_F(a) \rightarrow \bullet\, \underline{acc}(a) \qquad \text{and} \qquad \Box \forall a.\, acc_S(a) \rightarrow \neg \bullet\, \underline{acc}(a)\,.$$

The assumptions for mgr_S and mgr_F are similar and we omit them.

Our formalization of the policy that whenever a report is published, it must be published by an accountant and the report must be approved by her manager within at most 10 time units prior to publication is now given by the formula

$$\Box \forall a.\, \forall f.\, publish(a, f) \rightarrow \underline{acc}(a) \wedge \blacklozenge_{[0,11)}\, \exists m.\, \underline{mgr}(m, a) \wedge approve(m, f)\,. \quad \text{(P1)}$$

Note that the state predicates $\underline{acc}$ and $\underline{mgr}$ can change over time and that such changes are accounted for in our MFOTL formalization of this security policy. In particular, at the time point where m approves the report f, the formula (P1) requires that m is a's manager. However, m need no longer be a's manager when a publishes f, although a must be an accountant at that time point.

The resulting monitor for (P1) can be used in an offline setting, e.g., to read log files and report policy violations. When the monitor is built into a policy decision point, it can also be used, in this case, for policy enforcement.

3.2 Transaction Requirements

Our next example is a compliance policy for a banking system that processes customer transactions. The requirements stem from anti-money laundering regulations such as the Bank Secrecy Act [1] and the USA Patriot Act [2].

Signature. Let S be the signature (C, R, ι), with $C := \{th\}$, $R := \{trans, auth, report\}$, and $\iota(trans) := 3$, $\iota(auth) := 2$, and $\iota(report) := 1$. The ternary predicate *trans* represents the execution of a transaction of some customer transferring a sum of money. The binary predicate *auth* denotes the authorization of a transaction by some employee. Finally, the unary predicate *report* represents the situation where a transaction is reported as suspicious.

Formalization. We first formalize that executed transactions t of any customers c must be reported within at most 5 days if the transferred money a exceeds a given threshold *th*.

$$\Box \forall c.\, \forall t.\, \forall a.\, trans(c, t, a) \wedge th \prec a \rightarrow \Diamond_{[0,6)}\, report(t)\,. \quad \text{(P2)}$$

Moreover, transactions that exceed the threshold must be authorized by some employee e prior to execution.

$$\Box \forall c.\, \forall t.\, \forall a.\, trans(c, t, a) \wedge th \prec a \rightarrow \blacklozenge_{[2,21)}\, \exists e.\, auth(e, t)\,. \quad \text{(P3)}$$

Here we require that the authorization takes place at least 2 days and at most 20 days before executing the transaction. Our last requirement concerns the transactions of a customer that has previously made transactions that were classified as suspicious. Namely, every executed transaction t of a customer c,

who has within the last 30 days been involved in a suspicious transaction t', must be reported as suspicious within 2 days.

$$\Box \forall c.\, \forall t.\, \forall a.\, \mathit{trans}(c,t,a) \wedge \big(\blacklozenge_{[0,31)}\, \exists t'.\, \exists a'.\, \mathit{trans}(c,t',a') \wedge \Diamond_{[0,6)}\, \mathit{report}(t') \big) \rightarrow \Diamond_{[0,3)}\, \mathit{report}(t)\,. \tag{P4}$$

3.3 Experimental Evaluation

We implemented a Java prototype of the monitoring algorithm and evaluated its performance on the above policies. As input data, we synthetically generate finite prefixes of temporal structures, as this allows us to study the algorithm's performance under different parameter settings. Namely, for each formula, we synthesize finite prefixes of temporal structures over the formula's signature by drawing the time stamps and the elements of the relations from predefined sample spaces using a discrete uniform distribution. We restrict ourselves to relational structures with singleton relations that also satisfy the given well-formedness assumptions. To assess the monitor's long-run performance, we then conduct a steady-state analysis [35], which is a standard method for estimating the behavior of non-terminating processes in the limit. For more information on our experimental setup, see [9].

Table 1 summarizes our experimental results using a 1.4 GHz dual core computer with 3 GBytes of RAM. The size of the sample space for the m different kinds of data, e.g., managers, accountants, and files, is denoted by $\Omega_{n_1 \times \cdots \times n_m}$. The sample space for time stamps is chosen so that the different lengths of the generated temporal structures simulate scenarios with the (approximate) event frequencies $110, 220, \ldots, 550$, i.e., the number of structures associated with each time point that approximately occur in the time window specified by the metric temporal operators of the given formula. We measure the following aspects. (1) *ipt* denotes the steady-state mean incremental processing times, in milliseconds. The incremental processing time is the time the monitor needs to construct and update the auxiliary relations in one loop iteration. (2) *sc* denotes a point estimate of the steady-state mean space consumption, where the actual average space consumption lies in the specified interval with a probability of 95%. We measured the monitor's space consumption as the sum of the cardinalities of the auxiliary relations. (3) *omax* denotes the maximal space consumption that we observed in our experiments. Finally, (4) *radom* denotes the average of the cardinalities of the relevant active domains[4] after the warm-up phase.

[4] The *relevant active domain* with respect to a time point is the set of data elements of the temporal structure that appear in the relations in the formula's time window at the time point. Although these cardinalities are only a rough complexity measure for the processed input prefix, they help us judge the monitor's performance better than more simplistic measures like the cardinality of the active domain of the processed prefix or the length of the prefix. In particular, the cardinalities of the relevant active domains relate the incremental update times and the cardinalities of the auxiliary relations to the input prefix of a temporal structure with respect to the formula to be monitored. The elements that do not occur in the relevant active domain for a time point are irrelevant for detecting policy violations at that time point.

Table 1. Experimental results of the steady-state analysis

		event frequency					
formula	aspect	110	220	330	440	550	sample space
(P1)	*ipt*	14.1	21.8	26.0	37.7	39.4	$\Omega_{20\times20\times2000}$
	sc	672±70.5	1,267±135.2	1,857±200.3	2,442±265.4	3,024±331.2	
	omax	1,208	2,155	3,006	3,988	4,884	
	radom	281	477	661	818	950	
(P2)	*ipt*	7.0	13.1	17.9	21.0	29.6	$\Omega_{1000\times25000\times2}$
	sc	353±4.4	700±8.7	1,044±12.0	1,386±15.2	1,725±20.7	
	omax	2,135	3,959	5,172	7,377	8,714	
	radom	404	762	1,098	1,422	1,726	
(P3)	*ipt*	1.7	2.8	3.7	4.8	10.4	$\Omega_{1000\times25000\times2\times200}$
	sc	119±1.3	235±2.6	350±3.9	465±5.0	579±5.6	
	omax	158	282	412	545	659	
	radom	492	893	1,252	1,583	1,893	
(P4)	*ipt*	2.2	3.5	4.7	6.0	7.6	$\Omega_{1000\times25000\times2}$
	sc	140±2.8	405±9.0	801±19.1	1,334±32.2	1,994±47.8	
	omax	723	1,270	2,242	3,302	4,360	
	radom	404	762	1,098	1,422	1,726	

The results of our experiments, depicted in Table 1, predict low space consumption and running times of the monitoring algorithm in the long run. This suggests that we can monitor realistic policies with manageable overhead. Moreover, the monitoring algorithm scales well with respect to the event frequency: the growth rates of all four aspects measured are approximately linear with respect to the event frequency.

Our results also shed light on the relationship between formula structure and monitoring efficiency. The state predicates used in (P1) result in additional temporal subformulae and hence increased space consumption and processing time. Moreover, the maximal observed space consumption is close to the estimated steady-state mean space consumption for formulae only referring to the past. For formulae containing future operators, i.e. (P2) and (P4), these values differ up to a factor of 6 since the monitoring algorithm delays the policy check at time points when it depends on future events. The information about the current time point must be stored in auxiliary relations until this check is performed.

4 Related Work

Temporal logics are widely applicable in computing since they allow one to formally and naturally express system properties and they can be handled algorithmically. For instance, in system verification, the propositional temporal logics LTL, CTL, and PSL are widely used [16,42,50]. Linear-time temporal logics like first-order extensions of LTL and different real-time extensions [4] have also been used to formalize [8,19,24,28,29,31,51] and to reason about [8,14,19,20] system policies. However, reasoning about policies has been mostly carried out in just the propositional setting [8,20]. For instance, in [8], policy consistency is reduced to checking whether an LTL formula is satisfiable and verification techniques for LTL are proposed for checking runtime compliance. This kind of reasoning is inadequate for systems with unboundedly many users or data elements. Note that

although a system has only finitely many users at each time point, the number of users over time is usually unbounded.

In the domain of security and compliance checking, bounds on the number of users or data elements are usually unrealistic. Hence most monitoring algorithms, e.g. [11,17,18,21,23,30,34,40,44], are of limited use in this domain. The rule-based monitoring approach implemented in the closely related EAGLE [6] and RuleR [7] frameworks partially overcomes this limitation. There, properties are given as so-called temporal equations, which can have parameters referring to data that are instantiated during monitoring. EAGLE's rule-based approach has been used in [19] to monitor regulations, where one distinguishes between provisions and obligations and where regulations can refer to other regulations. Analogous to the use of parametric temporal equations in EAGLE and RuleR, the monitoring algorithm from [41,43] for auditing log files instantiates the parameters occurring in the given temporal formula during the monitoring process. Roughly speaking, such instantiations create propositions on demand and the number of propositions can be unbounded. These instantiations can also be seen as a restricted form of existential quantification, where variables are assigned to values that appear at the current position of the input trace.

The linear-time temporal logic used for monitoring in [25,26] directly supports quantification. However, quantified variables only range over elements that appear at the current position of the input trace. Similar to [6,7,43], quantification is handled by instantiations. In contrast, our monitoring algorithm does not create propositions at runtime. Instead it creates auxiliary relations for the temporal subformulae of the given MFOTL formula. Our monitoring algorithm thereby handles more general existential and universal quantification; however, formulae must be domain independent. A simple, albeit artificial, example is the formula $\Box\, \exists x.\, (\bigcirc\, p(x)) \wedge \neg q(x)$ whose negation is $\Diamond\, \forall x.\, (\bigcirc\, p(x)) \rightarrow q(x)$, which is in our monitorable fragment. However, elements $a \in |\bar{\mathcal{D}}|$ for which $a \in p^{\mathcal{D}_{i+1}}$ holds, need not appear at the current time point i. The monitoring approach in [48] is similar to the one in [25] but instead of using a tableau construction as in [25], it uses so-called parametric alternating automata, which are instantiated during runtime. Other differences to our monitoring algorithm are that the monitoring algorithms in [25,48] do not handle past operators and future operators need not be bounded.

Our monitoring algorithm is based on Chomicki's monitor for checking integrity constraints on temporal databases [13]. It extends and improves Chomicki's monitor by supporting bounded future operators and by simplifying and optimizing the incremental update constructions for the metric operators. Moreover, when using automatic structures, no syntactic restrictions on the MFOTL formula to domain-independent queries are necessary. Other monitoring algorithms for temporal databases are given in [38,46]. Both of these algorithms support only future operators and neither handles arbitrary quantifier alternation. Processing database streams is also related to monitoring and compliance checking. However, query languages like CQL [5] are less expressive temporally.

What they usually provide instead are operators for manipulating sequences, for example, transforming streams into relations and vice versa.

In this paper, our focus is on monitoring for compliance checking, rather than policy enforcement [37,45]. Enforcement is more difficult as it may necessitate changing future actions or predicting when existing actions have consistent extensions. It is also complicated by distribution, as a monitor may be able to observe events, but not necessarily control them.

5 Conclusions

We have given an overview of some of the ideas behind our approach to runtime monitoring using an expressive fragment of a metric first-order temporal logic. We have also given examples illustrating how policies can be formalized and we have analyzed the monitor's resource requirements.

Of course, our approach is not a panacea. Policies outside the scope of MFOTL include those for which no domain-independent formalization exists or those requiring a more expressive logic. An example of the latter is the requirement *a report must be filed within 3 days when all transactions of a trader over the last week sum up to more than $50 million*, involving the aggregation operator for summation. Similarly, our experiments indicate that the monitoring algorithm does not handle all policies equally well as a policy's syntactic form may influence monitoring efficiency. In general, for monitoring those properties formalizable in MFOTL, there may be more efficient, specialized algorithms than ours. Despite these limitations, MFOTL appears to sit in the sweet spot between expressivity and complexity: it is a large hammer, applicable to many problems, and has acceptable runtime performance.

We have indicated that our monitors can be used in some cases for policy enforcement. We plan to explore how this can best be done and to compare the performance with competing approaches. We would also like to carry out concrete case studies in the application domains presented in this paper.

References

1. Bank Secrecy Act of 1970. 31 USC 5311-5332 and 31 CFR 103 (1970)
2. USA Patriot Act of 2001. Public Law 107-56, HR 3162 RDS (2001)
3. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases. Addison-Wesley Longman Publishing Co., Inc., Boston (1995)
4. Alur, R., Henzinger, T.A.: Logics and models of real time: A survey. In: Huizing, C., de Bakker, J.W., Rozenberg, G., de Roever, W.-P. (eds.) REX 1991. LNCS, vol. 600, pp. 74–106. Springer, Heidelberg (1992)
5. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. VLDB Journal 15(2), 121–142 (2006)
6. Barringer, H., Goldberg, A., Havelund, K., Sen, K.: Rule-based runtime verification. In: Steffen, B., Levi, G. (eds.) VMCAI 2004. LNCS, vol. 2937, pp. 44–57. Springer, Heidelberg (2004)

7. Barringer, H., Rydeheard, D.E., Havelund, K.: Rule systems for run-time monitoring: From Eagle to RuleR. J. Logic Comput. (to appear)
8. Barth, A., Datta, A., Mitchell, J.C., Nissenbaum, H.: Privacy and contextual integrity: Framework and applications. In: Proc. of the 2006 IEEE Symposium on Security and Privacy, pp. 184–198. IEEE Computer Society, Los Alamitos (2006)
9. Basin, D., Klaedtke, F., Müller, S.: Monitoring security policies with metric first-order temporal logic. In: 15th ACM Symposium on Access Control Models and Technologies (SACMAT) (accepted for publication)
10. Basin, D., Klaedtke, F., Müller, S., Pfitzmann, B.: Runtime monitoring of metric first-order temporal properties. In: Proc. of the 28th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), Dagstuhl Seminar Proc., pp. 49–60 (2008)
11. Bauer, A., Leucker, M., Schallhart, C.: Monitoring of real-time properties. In: Arun-Kumar, S., Garg, N. (eds.) FSTTCS 2006. LNCS, vol. 4337, pp. 260–272. Springer, Heidelberg (2006)
12. Blumensath, A., Grädel, E.: Finite presentations of infinite structures: Automata and interpretations. Theory Comput. Syst. 37(6), 641–674 (2004)
13. Chomicki, J.: Efficient checking of temporal integrity constraints using bounded history encoding. ACM Trans. Database Syst. 20(2), 149–186 (1995)
14. Chomicki, J., Lobo, J.: Monitors for history-based policies. In: Sloman, M., Lobo, J., Lupu, E.C. (eds.) POLICY 2001. LNCS, vol. 1995, pp. 57–72. Springer, Heidelberg (2001)
15. Chomicki, J., Niwiński, D.: On the feasibility of checking temporal integrity constraints. J. Comput. Syst. Sci. 51(3), 523–535 (1995)
16. Clarke, E.M., Emerson, E.A.: Design and synthesis of synchronization skeletons using branching-time temporal logic. In: Kozen, D. (ed.) Logic of Programs 1981. LNCS, vol. 131, pp. 52–71. Springer, Heidelberg (1982)
17. d'Amorim, M., Roşu, G.: Efficient monitoring of ω-languages. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 364–378. Springer, Heidelberg (2005)
18. D'Angelo, B., Sankaranarayanan, S., Sánchez, C., Robinson, W., Finkbeiner, B., Sipma, H.B., Mehrotra, S., Manna, Z.: LOLA: Runtime monitoring of synchronous systems. In: Proc. of the 12th International Symposium on Temporal Representation and Reasoning (TIME), pp. 166–174. IEEE Computer Society, Los Alamitos (2005)
19. Dinesh, N., Joshi, A., Lee, I., Sokolsky, O.: Checking traces for regulatory conformance. In: Leucker, M. (ed.) RV 2008. LNCS, vol. 5289, pp. 86–103. Springer, Heidelberg (2008)
20. Dougherty, D.J., Fisler, K., Krishnamurthi, S.: Obligations and their interaction with programs. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 375–389. Springer, Heidelberg (2007)
21. Drusinsky, D.: On-line monitoring of metric temporal logic with time-series constraints using alternating finite automata. Journal of Universal Computer Science 12(5), 482–498 (2006)
22. Fagin, R.: Horn clauses and database dependencies. J. ACM 29(4), 952–985 (1982)
23. Giannakopoulou, D., Havelund, K.: Automata-based verification of temporal properties on running programs. In: Proc. of the 16th IEEE International Conference on Automated Software Engineering (ASE), pp. 412–416. IEEE Computer Society, Los Alamitos (2001)

24. Giblin, C., Liu, A.Y., Müller, S., Pfitzmann, B., Zhou, X.: Regulations expressed as logical models (REALM). In: Proc. of the 18th Annual Conference on Legal Knowledge and Information Systems (JURIX). Frontiers Artificial Intelligence Appl., vol. 134, pp. 37–48. IOS Press, Amsterdam (2005)
25. Hallé, S., Villemaire, R.: Runtime monitoring of message-based workflows with data. In: Proc. of the 12th International IEEE Enterprise Distributed Object Computing Conference (EDOC), pp. 63–72. IEEE Computer Society, Los Alamitos (2008)
26. Hallé, S., Villemaire, R.: Browser-based enforcement of interface contracts in web applications with BeepBeep. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 648–653. Springer, Heidelberg (2009)
27. Havelund, K., Roşu, G.: Efficient monitoring of safety properties. Int. J. Softw. Tools Technol. Trans. 6(2), 158–173 (2004)
28. Hilty, M., Basin, D., Pretschner, A.: On obligations. In: di Vimercati, S.d.C., Syverson, P.F., Gollmann, D. (eds.) ESORICS 2005. LNCS, vol. 3679, pp. 98–117. Springer, Heidelberg (2005)
29. Hilty, M., Pretschner, A., Basin, D., Schaefer, C., Walter, T.: A policy language for distributed usage control. In: Biskup, J., López, J. (eds.) ESORICS 2007. LNCS, vol. 4734, pp. 531–546. Springer, Heidelberg (2007)
30. Håkansson, J., Jonsson, B., Lundqvist, O.: Generating online test oracles from temporal logic specifications. Int. J. Softw. Tools Technol. Trans. 4(4), 456–471 (2003)
31. Janicke, H., Cau, A., Zedan, H.: A note on the formalisation of UCON. In: Proc. of the 12th ACM Symposium on Access Control Models and Technologies (SACMAT), pp. 163–168. ACM Press, New York (2007)
32. Khoussainov, B., Nerode, A.: Automatic presentations of structures. In: Leivant, D. (ed.) LCC 1994. LNCS, vol. 960, pp. 367–392. Springer, Heidelberg (1995)
33. Koymans, R.: Specifying real-time properties with metric temporal logic. Real-Time Syst. 2(4), 255–299 (1990)
34. Kristoffersen, K.J., Pedersen, C., Andersen, H.R.: Runtime verification of timed LTL using disjunctive normalized equation systems. Elec. Notes Theo. Comput. Sci. 89(2), 1–16 (2003)
35. Law, A.M.: Simulation, Modeling & Analysis, 4th edn. McGraw-Hill, New York (2007)
36. Lichtenstein, O., Pnueli, A., Zuck, L.D.: The glory of the past. In: Parikh, R. (ed.) Logic of Programs 1985. LNCS, vol. 193, pp. 196–218. Springer, Heidelberg (1985)
37. Ligatti, J., Bauer, L., Walker, D.: Edit automata: enforcement mechanisms for run-time security policies. Int. J. Inf. Secur. 4(1-2), 2–16 (2005)
38. Lipeck, U.W., Saake, G.: Monitoring dynamic integrity constraints based on temporal logic. Information Systems 12(3), 255–269 (1987)
39. Maler, O., Nickovic, D., Pnueli, A.: From MITL to timed automata. In: Asarin, E., Bouyer, P. (eds.) FORMATS 2006. LNCS, vol. 4202, pp. 274–289. Springer, Heidelberg (2006)
40. 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)
41. Olivain, J., Goubault-Larrecq, J.: The Orchids intrusion detection tool. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 286–290. Springer, Heidelberg (2005)

42. Pnueli, A.: The temporal logic of programs. In: Proc. of the 18th IEEE Symposium on Foundations of Computer Science (FOCS), pp. 46–57. IEEE Computer Society, Los Alamitos (1977)
43. Roger, M., Goubault-Larrecq, J.: Log auditing through model-checking. In: Proc. of the 14th IEEE Computer Security Foundations Workshop (CSFW), pp. 220–234. IEEE Computer Society, Los Alamitos (2001)
44. Roşu, G., Havelund, K.: Rewriting-based techniques for runtime verification. Automat. Softw. Eng. 12(2), 151–197 (2005)
45. Schneider, F.B.: Enforceable security policies. ACM Trans. Inform. Syst. Secur. 3(1), 30–50 (2000)
46. Sistla, A.P., Wolfson, O.: Temporal triggers in active databases. IEEE Trans. Knowl. Data Eng. 7(3), 471–486 (1995)
47. Sokolsky, O., Sammapun, U., Lee, I., Kim, J.: Run-time checking of dynamic properties. Elec. Notes Theo. Comput. Sci. 144(4), 91–108 (2006)
48. Stolz, V.: Temporal assertions with parameterized propositions. J. Logic Comput. (to appear)
49. Thati, P., Roşu, G.: Monitoring algorithms for metric temporal logic specifications. Elec. Notes Theo. Comput. Sci. 113, 145–162 (2005)
50. Vardi, M.Y.: From philosophical to industrial logics. In: Ramanujam, R., Sarukkai, S. (eds.) ICLA 2009. LNCS (LNAI), vol. 5378, pp. 89–115. Springer, Heidelberg (2009)
51. Zhang, X., Parisi-Presicce, F., Sandhu, R., Park, J.: Formal model and policy specification of usage control. ACM Trans. Inform. Syst. Secur. 8(4), 351–387 (2005)

Retrofitting Legacy Code for Security
(Invited Talk)

Somesh Jha

University of Wisconsin, Madison, USA

Research in computer security has historically advocated Design for Security, the principle that security must be proactively integrated into the design of a system. While examples exist in the research literature of systems that have been designed for security, there are few examples of such systems deployed in the real world. Economic and practical considerations force developers to abandon security and focus instead on functionality and performance, which are more tangible than security. As a result, large bodies of legacy code often have inadequate security mechanisms. Security mechanisms are added to legacy code on-demand using ad hoc and manual techniques, and the resulting systems are often insecure.

This talk advocates the need for techniques to retrofit systems with security mechanisms. In particular, it focuses on the problem of retrofitting legacy code with mechanisms for authorization policy enforcement. It introduces a new formalism, called fingerprints, to represent security-sensitive operations. Fingerprints are code templates that represent accesses to security-critical resources, and denote key steps needed to perform operations on these resources. This talk develops both fingerprint mining and fingerprint matching algorithms.

Fingerprint mining algorithms discover fingerprints of security-sensitive operations by analyzing source code. This talk presents two novel algorithms that use dynamic program analysis and static program analysis, respectively, to mine fingerprints. The fingerprints so mined are used by the fingerprint matching algorithm to statically locate security-sensitive operations. Program transformation is then employed to statically modify source code by adding authorization policy lookups at each location that performs a security-sensitive operation.

These techniques have been applied to three real-world systems. These case studies demonstrate that techniques based upon program analysis and transformation offer a principled and automated alternative to the ad hoc and manual techniques that are currently used to retrofit legacy software with security mechanisms. Time permitting, we will talk about other problems in the context of retrofitting legacy code for security. I will also indicate where ideas from model-checking have been used in this work.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, p. 19, 2010.

Quantitative Information Flow: From Theory to Practice?

Pasquale Malacaria

School of Electronic Engineering and Computer Science
Queen Mary University of London

Most computational systems share two basic properties: first they process information, second they allow for observations of this processing to be made. For example a program will typically process the inputs and allows its output to be observed on a screen. In a distributed system each unit processes information and will allow some observation to be made by the environment or other units, for example by message passing. In an election voters cast their vote, officials count the votes and the public can observe the election result.

The general aim of Quantitative Information Flow (QIF) [4,5] is to measure the amount of information about a source of information that can be gained by observations on some related system component. An application in the context of programming languages is for example to measure leakage of confidential information in programs.

The simplest example motivating QIF is a password checking program where secret data (stored in the high variable `h`) can be deduced by observing the final value of the low variable `l`:

$$\texttt{if } (\texttt{h} == \texttt{l})\ \texttt{l} = 0\ \texttt{else}\ \texttt{l} = 1$$

The above program is insecure and the aim of QIF is to measure its insecurity.

Further examples of this scenario is code handling medical records, code querying databases, arithmetic operations within a cryptographic package, RFID tag identification etc. In these examples we expect sound code to leak as little as possible. There are also positive applications of QIF, where the analysis should return high leakage; examples of this scenario are programs computing best fit representation for data, hash coding, image processing code, strategy game solving programs etc.

In both scenarios would be very useful to have reasoning techniques and tools, possibly automatic tools, able to measure how much information is leaked.

QIF attempts to answer a set of questions related to leakage, for example are we interested in the total possible amount leaked, or only if it leaks below or above a certain threshold? clearly the former question is in general harder than the latter because for the latter we can stop measuring leakage once reached the threshold. Also is the ration leaked or the amount itself the information we want? for example a 3 bits leak is one tenth for a 30 bits secret but is one hundred percent for a 3 bits secret.

The nature of what is leaked is also very important, typically does it leak always the same information? for example there is a very big difference if the

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 20–22, 2010.

code leaks always the same bit when it is run or if leaks a different bit every time is run. Other important questions are: at what rate, i.e. how fast, is information leaked? and what's the maximum leakage (the channel capacity) over all possible choice of probabilities for the inputs?

The fundamental question has to be what is exactly that we want to use as a measure [13].

The basic albeit crude unit of measure in the analysis is the number of states of the source of information that can be distinguished by the observations. Classical non-interference [8] amounts to not to be able to make any distinction by looking at the observations, whereas total leakage corresponds to be able to make all possible distinctions. The tricky bit is, of course, what lies in between these two extreme.

In the past years a number of works have refined this idea and provided a solid theoretical background for QIF [5,10,13]. The theory is based on Information Theoretical terms, the main being Shannon's entropy but also Renyi's entropies and guessability measures have been shown to be relevant. The Information Theoretical approach has been shown to be powerful and able to provide natural, intuitive and precise reasoning principles for program constructs including loops [9,10]. The theory has also been fruitfully applied to the analysis of anonymity protocols, where the same framework has been used to measure the loss of anonymity [2,3].

More recent works have investigated automation of such ideas, and verification techniques like abstract interpretation, bounded model checkers and SAT solvers have been applied in this context [1,7,11,12]. Implementation of the quantitative aspects of the analysis presents however a number of challenges [14], the main being scalability and handling of data structures.

Some of these challenges could probably be solved by appropriate abstraction techniques, some may be better addressed by statistical means [6].

The talk will give an overview of the field, the basic ideas, reasoning principles and problems.

References

1. Backes, M., Köpf, B., Rybalchenko, A.: Automatic Discovery and Quantification of Information Leaks. In: Proc. 30th IEEE Symposium on Security and Privacy S& P '09 (2009)
2. Chatzikokolakis, K., Palamidessi, C., Panangaden, P.: Anonymity protocols as noisy channels. Information and Computation 206(2-4), 378–401 (2008)
3. Chen, H., Malacaria, P.: Quantifying Maximal Loss of Anonymity in Protocols. In: Proceedings ACM Symposium on Information, Computer and Communication Security (2009)
4. Clark, D., Hunt, S., Malacaria, P.: A static analysis for quantifying information flow in a simple imperative language. Journal of Computer Security 15(3) (2007)
5. Clark, D., Hunt, S., Malacaria, P.: Quantitative information flow, relations and polymorphic types. Journal of Logic and Computation, Special Issue on Lambda-calculus, type theory and natural language 18(2), 181–199 (2005)

6. Kpf, B., Rybalchenko, A.: Approximation and Randomization for Quantitative Information-Flow Analysis. In: Proceedings of Computer Security Foundations Symposium 2010 (2010)
7. Heusser, J., Malacaria, P.: Applied Quantitative Information Flow and Statistical Databases. In: Proceedings of Workshop on Formal Aspects in Security and Trust, FAST 2009 (2009)
8. Goguen, J.A., Meseguer, J.: Security policies and security model. In: Proceedings of the 1982 IEEE Computer Society Symposium on Security and Privacy (1982)
9. Malacaria, P.: Assessing security threats of looping constructs. In: Proc. ACM Symposium on Principles of Programming Language (2007)
10. Malacaria, P.: Risk Assessment of Security Threats for Looping Constructs. Journal Of Computer Security 18(2) (2010)
11. McCamant, S., Ernst, M.D.: Quantitative information flow as network flow capacity. In: PLDI 2008, Proceedings of the ACM SIGPLAN 2008, Conference on Programming Language Design and Implementation, Tucson, AZ, USA (2008)
12. Mu, C., Clark, D.: An Abstraction Quantifying Information Flow over Probabilistic Semantics. In: Workshop on Quantitative Aspects of Programming Languages (QAPL), ETAPS (2009)
13. 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)
14. Yasuoka, H., Terauchi, T.: Quantitative Information Flow - Verification Hardness and Possibilities. In: Proceedings of Computer Security Foundations Symposium (2010)

Memory Management in Concurrent Algorithms (Invited Talk)

Maged M. Michael

IBM Thomas J. Watson Research Center

Many shared memory concurrent algorithms involve accessing dynamic nodes of shared structures optimistically, where a thread may access dynamic nodes while they are being updated by concurrent threads. Optimistic access is often necessary to enable non-blocking progress, and it is desirable for increasing concurrency and reducing conflicts among concurrent operations.

Optimistic access to dynamic nodes and structures makes the memory management of such nodes one of the most complex aspects of concurrent algorithms. For example, a thread operating on a shared dynamic structure may access a dynamic node after it was removed from the structure by another thread, and possibly even after it was reclaimed for unrelated reuse. Unless the algorithm and the memory management method it employs are carefully designed to handle such situations correctly, they can lead to various errors such as memory access violations, return of incorrect results, and/or corruption of shared data.

The main purpose of memory management in the context of concurrent algorithms is to balance the goal of enabling and maximizing the flexible reuse of dynamic nodes that are no longer needed, with the goal of maximizing the flexibility of safe access to dynamic nodes.

The verification of a concurrent algorithm is incomplete without taking into account the algorithm's memory management properties. Some such properties need to be verified explicitly, as they are not necessarily covered by conventional safety and progress properties, such as linearizability and deadlock-freedom. Understanding the subtleties of memory management properties of concurrent algorithms and the relations between these properties can help make verification of concurrent algorithms more effective and complete.

The talk discusses memory management problems and issues in concurrent algorithms, including the tightly-related ABA problem, categories of memory management and ABA-prevention solutions, types of memory access properties of concurrent algorithms, memory bounds, progress properties of memory management methods, and relations among these various properties.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, p. 23, 2010.

ABC: An Academic Industrial-Strength Verification Tool

Robert Brayton and Alan Mishchenko

EECS Department, University of California, Berkeley, CA 94720, USA
{brayton,alanmi}@eecs.berkeley.edu

Abstract. ABC is a public-domain system for logic synthesis and formal verification of binary logic circuits appearing in synchronous hardware designs. ABC combines scalable logic transformations based on And-Inverter Graphs (AIGs), with a variety of innovative algorithms. A focus on the synergy of sequential synthesis and sequential verification leads to improvements in both domains. This paper introduces ABC, motivates its development, and illustrates its use in formal verification.

Keywords: Model checking, equivalence checking, logic synthesis, simulation, integrated sequential verification flow.

1 Introduction

Progress in both academic research and industrial products critically depends on the availability of cutting-edge open-source tools in the given domain of EDA. Such tools can be used for benchmarking, comparison, and education. They provide a shared platform for experiments and can help simplify the development of new algorithms. Equally important for progress is access to real industrial-sized benchmarks.

For many years, the common base for research in logic synthesis has been SIS, a synthesis system developed by our research group at UC Berkeley in 1987-1991. Both SIS [35] and its predecessor MIS [8], pioneered multi-level combinational logic synthesis and became trend-setting prototypes for a large number of synthesis tools developed by industry.

In the domain of formal verification, a similar public system has been VIS [9], started at UC Berkeley around 1995 and continued at the University of Colorado, Boulder, and University of Texas, Austin. In particular, VIS features the latest algorithms for implicit state enumeration [15] with BDDs [11] using the CUDD package [36].

While SIS reached a plateau in its development in the middle 90's, logic representation and manipulation methods continued to be improved. In the early 2000s, And-Inverter Graphs (AIGs) emerged as a new efficient representation for problems arising in formal verification [22], largely due to the published work of A. Kuehlmann and his colleagues at IBM.

In that same period, our research group worked on a multi-valued logic synthesis system, MVSIS [13]. Aiming to find better ways to manipulate multi-valued relations, we experimented with new logic representations, such as AIGs, and found that, in addition to their use in formal verification, they can replace more-traditional

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 24–40, 2010.

representations in logic synthesis. As a result of our experiments with MVSIS, we developed a methodology for tackling problems, which are traditionally solved with SOPs [35] and BDDs [37], using a combination of random/guided simulation of AIGs and Boolean satisfiability (SAT) [25].

Based on AIGs as a new efficient representation for large logic cones, and SAT as a new way of solving Boolean problems, in the summer 2005, we switched from multi-valued methods in MVSIS to binary AIG-based synthesis methods. The resulting CAD system, ABC, incorporates the best algorithmic features of MVSIS, while supplementing them with new findings.

One such finding is a novel method for AIG-based logic synthesis that replaced the traditional SIS logic synthesis flow, which was based on iterating elimination, substitution, kerneling, don't-care-based simplification, as exemplified by SIS scripts, *script.algebraic* and *script.rugged*. Our work on AIGs was motivated by fast compression of Boolean networks in formal verification [5]. We extended this method to work in synthesis, by making it delay-aware and replacing two-level structural matching of AIG subgraphs with functional matching of the subgraphs based on enumeration of 4-input cuts [26].

It turned out that the fast AIG-based optimizations could be made even more efficient by applying them to the network many times. Doing so with different parameter settings led to results in synthesis comparable or better than those of SIS, while requiring much less memory and runtime. Also this method is conceptually simpler than the SIS optimization flow, saving months of human-effort in code development and tuning. The savings in runtime/memory led to the increased scalability of ABC, compared to SIS. As a result, ABC can work on designs with millions of nodes, while SIS does not finish on these designs after many hours, and even if it finishes, the results are often inferior to those obtained by the fast iterative computations in ABC.

The next step in developing ABC was to add an equivalence checker for verifying the results of synthesis, both combinational and sequential [29]. Successful equivalence checking motivated experiments with model checking, because both types of verification work on a miter circuit and have the common goal of reducing it to the constant 0. To test this out, we submitted an equivalence checker in ABC to the hardware model checking competition at CAV 2008, winning in two out of three categories.

Working on both sequential synthesis and verification has allowed us to leverage the latest results in both domains and observe their growing synergy. For example, the ability to synthesize large problems and show impressive gains spurs development of equally scalable equivalence checking methods, while the ability to scalably verify sequential equivalence problems spurs the development, use, and acceptance of aggressive sequential synthesis. In ABC, similar concepts are used in both synthesis and verification: AIGs, rewriting, SAT, sequential SAT sweeping, retiming, interpolation, etc.

This paper provides an overview of ABC, lists some of the ways in which verification ideas have enriched synthesis methods, shows how verification is helped by constraining or augmenting sequential synthesis, and details how various algorithms have been put together to create a fairly powerful model checking engine that can rival some commercial offerings. We give an example of the verification flow applied to an industrial model checking problem.

The rest of the paper is organized as follows. Section 2 introduces the basic terminology used in logic synthesis and verification. Section 3 describes combinational and sequential AIGs and their advantages over traditional representations. Section 4 discusses the duality of synthesis and verification. Section 5 gives a case study of an efficient AIG implementation, complete with experimental results. Section 6 describes both the synthesis and verification flows in ABC and provides an example of the verification flow applied to an industrial model checking problem. Section 7 concludes the paper and sketches some on-going research.

2 Background

2.1 Boolean Network

A *Boolean network* is a directed acyclic graph (DAG) with nodes corresponding to logic gates and directed edges corresponding to wires connecting the gates. The terms Boolean network, netlist, and circuit are used interchangeably in this paper. If the network is sequential, the memory elements are assumed to be D flip-flops with initial states.

A node *n* has zero or more *fanins*, i.e. nodes driving *n*, and zero or more *fanouts*, i.e. nodes driven by *n*. The *primary inputs* (PIs) are nodes without fanins in the current network. The *primary outputs* (POs) are a subset of nodes of the network. A *fanin (fanout) cone* of node *n* is a subset of all nodes of the network, reachable through the fanin (fanout) edges of the node.

2.2 Logic Synthesis

Logic synthesis transforms a Boolean network to satisfy some criteria, for example, reduce the number of nodes, logic levels, switching activity. *Technology mapping* deals with representing the logic in terms of a given set of primitives, such as standard cells or lookup tables.

Combinational logic synthesis involves changing the combinational logic of the circuit with no knowledge of its reachable states. As a result, the Boolean functions of the POs and register inputs are preserved for any state of the registers. In contrast, *sequential logic synthesis* preserves behavior on the reachable states and allows arbitrary changes on the unreachable states. Thus, after sequential synthesis, the combinational functions of the POs and register inputs may have changed, but the resulting circuit is sequentially-equivalent to the original.

2.3 Formal Verification

Formal verification tries to prove that the design is correct in some sense.

The two most common forms of formal verification are model checking and equivalence checking. *Model checking* of safety properties considers the design along with one or more properties and checks if the properties hold on all reachable states. *Equivalence checking* checks if the design after synthesis is equal to its initial version, called the *golden* model.

In modern verification flows, the circuit to be model-checked is transformed into a circuit called a model checking miter by supplementing the logic of the design with a monitor logic, which checks the correctness of the property. Similarly, in equivalence checking, the two circuits to be verified are transformed into an equivalence checking miter [7] derived by combining the pairs of inputs with the same names and feeding the pairs of outputs with the same names into EXOR gates, which are ORed to produce the single output of the miter.

In both property and equivalence checking, the miter is a circuit with the inputs of the original circuit and an output that produces value 0, if and only if the original circuit satisfies the property or if the two circuits produce identical output values under any input assignment (or, in sequential verification, under any sequence of input assignments, starting from the initial state).

The task of formal verification is to prove that the constructed miter always produces value 0. If synthesis alone does not solve the miter, the output can be asserted to be constant 1 and a SAT solver can be run on the resulting problem. If the solver returns "unsatisfiable", the miter is proved constant 0 and the property holds, or the original circuits are equivalent. If the solver returns "satisfiable", an assignment of the PIs leading to 1 at the output of the miter, called a counter-example, is produced, which is useful for debugging the circuit.

2.4 Verifiable Synthesis

An ultimate goal of a synthesis system is to produce good results in terms of area, power, speed, capability for physical implementation etc, while allowing an unbiased (independent) verification tool to prove that functionality is preserved. Developing verifiable synthesis methods is difficult because of the inherent complexity of the sequential verification problem [20].

One *verifiable sequential synthesis* is described in [29]. This is based on identifying pairs of sequentially-equivalent nodes/registers, that is groups of signals having the same or opposite values in all reachable states. Such equivalent nodes/registers can be merged without changing the sequential behavior of the circuit, often leading to substantial reductions, e.g. some parts of the logic can be discarded because they no longer affect the POs. This sequential synthesis technique is used extensively in ABC to reduce both designs and sequential miters.

3 And-Inverter Graphs

3.1 Combinational AIGs

A combinational *And-Invertor Graph* (AIG) is a Boolean network composed of two-input ANDs and inverters. To derive an AIG, the SOPs of the nodes in a logic network are factored, the AND and OR gates of the factored forms are converted into two-input ANDs and inverters using DeMorgan's rule, and these nodes are added to the AIG manager in a topological order. The *size* (*area*) of an AIG is the number of its nodes; the *depth* (*delay*) is the number of nodes on the longest path from the PIs to the POs. The goal of optimization by local transformations of an AIG is to reduce both area and delay.

Structural hashing of AIGs ensures that all constants are propagated and, for each pair of nodes, there is at most one AND node having them as fanins (up to a permutation). Structural hashing is performed by hash-table lookups when AND nodes are created and added to an AIG manager. Structural hashing was originally introduced for netlists of arbitrary gates in early IBM CAD tools [15] and was extensively used in formal verification [22]. Structural hashing can be applied on-the-fly during AIG construction, which reduces the AIG size. To reduce the number of AIG levels, the AIG is often *balanced* by applying the associative transform, $a(bc) = (ab)c$. Both structural hashing and balancing are performed in one topological traversal from the PIs and have linear complexity in the number of AIG nodes.

A *cut C* of a node *n* is a set of nodes of the network, called *leaves* of the cut, such that each path from a PI to *n* passes through at least one leaf. Node *n* is called the *root* of cut *C*. The cut *size* is the number of its leaves. A *trivial cut* of a node is the cut composed of the node itself. A cut is *K-feasible* if the number of nodes in the cut does not exceed *K*. A cut is *dominated* if there is another cut of the same node, which is contained, set-theoretically, in the given cut.

A *local* function of an AIG node *n*, denoted $f_n(x)$, is a Boolean function of the logic cone rooted in *n* and expressed in terms of the leaves, *x*, of a cut of *n*. The *global function* of an AIG node is its function in terms of the PIs of the network.

3.2 Sequential AIGs

Sequential AIGs extend combinational AIGs with technology-independent D-flip-flops with one input and one output, controlled by the same clock, omitted in the AIG representations.

We represent flip-flops in the AIG explicitly as additional PI/POs pairs. The PIs and register outputs are called *combinational inputs* (CIs) and the POs and register inputs are called *combinational outputs* (COs). The additional pairs of CI/CO nodes follow the regular PIs/POs, and are in one to one correspondence with each other. This representation of sequential AIGs differs from that used in [1] where latches are represented as attributes on AIG edges, similar to the work of Leiserson and Saxe [23].

The chosen representation of sequential AIGs allows us to work with the AIG manager as if it was a combinational AIG, and only utilize its sequential properties when sequential transformations are applied. For example, combinational AIG rewriting works uniformly on combinational and sequential AIGs, while sequential cleanup, which removes structurally equivalent flip-flops, exploits the fact that they are represented as additional PIs and POs. Sequential transformation, such as retiming, can add and remove latches as needed.

3.3 Distinctive Features of AIGs

Representing logic using networks containing two-input nodes is not new. In SIS, there is a command *tech_decomp* [35] generating a two-input AND/OR decomposition of the network. However, there are several important differences that make two-input node representation in the form of AIGs much more efficient that its predecessors in SIS:

- Structural hashing ensures that AIGs do not contain structurally identical nodes. For example, node $a \wedge b$ can only exist in one copy. When a new node is being created, the hash table is checked, and if such node already exists, the old node is returned. The on-the-fly structural hashing is very important in synthesis applications because, by giving a global view of the AIG, it finds, in constant time, simple logic sharing across the network.
- Representing inverters as edge attributes. This feature is borrowed from the efficient implementation of BDDs using complemented edges [36]. As a result, single-input nodes representing invertors and buffers do not have to be created. This saves memory and allows for applying DeMorgan's rule on-the-fly, which increases logic sharing.
- The AIG representation is uniform and fine-grain, resulting in a small, fixed amount of memory per node. The nodes are stored in one memory array in a topological order, resulting in fast, CPU-cache-friendly traversals. To further save memory, our AIG packages compute fanout information on demand, resulting in 50% memory reduction in most applications. Similar to the AIG itself, fanout information for arbitrary AIG structures can be represented efficiently using a constant amount of memory per node.

Fig. 1 shows a Boolean function and two of its structurally-different AIGs. The nodes in the graphs denote AND-gates, while the bubbles stand for complemented edges. The figure shows that the same completely-specified Boolean function can be represented by two structurally different AIGs, one with smaller size and larger depth, the other vice versa.

3.4 Comparing Logic Synthesis in SIS and in ABC

In terms of logic representation, the main difference between SIS and ABC, is that SIS works on a logic network whose nodes are represented using SOPs, while ABC works on an AIG whose nodes are two-input AND gates. A SIS network can be converted into an AIG by decomposing each node into two-input AND gates. For a deterministic decomposition algorithm, the resulting AIG is unique. However, the reverse transformation is not unique, because many logic networks can be derived from the same AIG by grouping AND gates in different ways. This constitutes the main difference between SIS and ABC.

SIS works on one copy of a logic network, defined by the current boundaries of its logic nodes, while ABC works on an AIG. A cut computed for an AND node in the AIG can be seen as a logic node. Since there are many cuts per logic node, the AIG can be seen as an implicit representation of many logic networks. When AIG rewriting is performed in ABC, a minimal representation is found among all decompositions of all structural cuts in the AIG, while global logic sharing is captured using a structural hashing table. Thus, ABC is more likely to find a smaller representation in terms of AIG nodes than SIS, which works on one copy of the logic network and performs only those transformations that are allowed by this network.

SIS and ABC use different heuristics for logic manipulation, so it is still possible that, for a particular network, SIS finds a better solution than ABC.

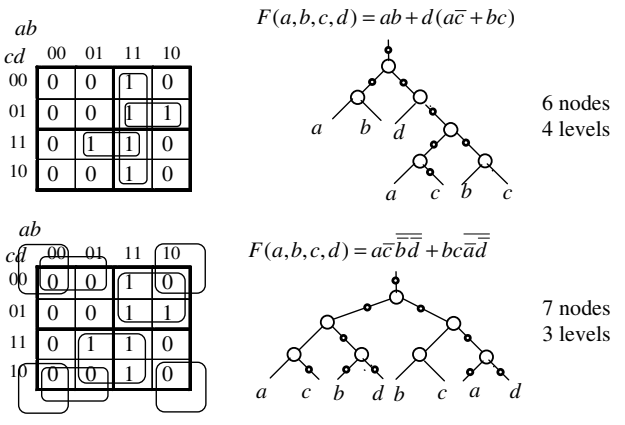


Fig. 1. Two different AIGs for a Boolean function

3.5 Advantages of AIGs summarized

The following properties of AIGs fascilitate development of robust applications in synthesis, mapping, and formal verification:

- AIGs unify the synthesis/mapping/verification by representing logic compactly and uniformly. The results of technology-independent synthesis are naturally expressed as an AIG. During technology mapping, the AIG is used as a subject graph annotated with cuts that are matched with LUTs or gates. At any time, verification can be performed by contructing a miter of the two synthesis snapshots represented as one AIG, handled by a complex AIG-based verification flow.
- Although AIG transformations are local, they are performed with a global view afforded by the structural hashing table. Because these computations are memory/runtime efficient, they can be iterated, leading to superior results, unmatched by a single application of a more global transform.
- An AIG can be efficiently duplicated, stored, and passed between calling applications as a memory buffer or compactly stored on disk in the AIGER format [4].

4 Synthesis-Verification Duality

Recent advances in formal verification and logic synthesis have made these fields increasingly interdependent, especially in the sequential domain [10].

In addition to algorithm migration (for example, AIG rewriting, SAT solving, interpolation came to synthesis from verification), hard verification problems challenge synthesis methods that are used to simplify them, while robust verification solutions enable more aggressive synthesis. For example, bold moves can be made in sequential synthesis by assuming something that seems likely to hold but cannot be proved easily. If the result can be verified (provided that sequential verification is

powerful enough), synthesis is over. Otherwise, different types of synthesis can be tried, for example, traditional or other risk-free synthesis. Preliminary experiments show potentially large gains in synthesis for industrial problems.

4.1 Known Synergies

In this section, we outline several aspects of combinational and sequential verification that benefit from synthesis.

Combinational equivalence checking (CEC) proves equivalence of primary outputs and register inputs after combinational synthesis. To this end, a combinational miter is constructed and solved using a set of integrated methods, including simulation, SAT solving, and BDD or SAT sweeping [22]. Running combinational synthesis on a miter during verification substantially improves the CEC runtime [27]. This is because synthesis quickly merges shallow equivalences and reduces the size of the miter, allowing difficult SAT calls go through faster.

A similar observation can be made about *retiming* [23]. If retiming has been applied during sequential synthesis, it is advantageous to apply *most-forward* retiming as one of the preprocessing steps during sequential verification. It can be shown that if during sequential synthesis only retiming was applied without changing the logic structure, then most forward retiming followed by an inductive register correspondence computation is guaranteed to prove sequential equivalence [21]. This observation is used in our verification tool, which allows the user to enable retiming as an intermediate step during sequential verification [29].

Yet another synthesis/verification synergy holds when induction is used to detect and merge sequentially equivalent nodes. The following result was obtained in [29]: if a circuit was synthesized using only k-step induction to find equivalent signals, then equivalence between the original and final circuits is guaranteed provable using k-step induction with the same k.

These results lead to the following rule of thumb which is used in our verification flow: if a transformation is applied during synthesis, it is often helpful (and necessary) to apply the same or more powerful transformations during verification.

5 Case Study: Developing a Fast Sequential Simulator for AIGs

Several applications suffer from the prohibitive runtime of a sequential gate-level simulator. For example, in formal verification, the simulator is used to quickly detect easy-to-disprove properties or as a way to compute simulation signatures of internal nodes proving their equivalence. The same sequential simulator is useful to estimate switching activity of registers and internal nodes. The pre-computed switching activity can direct transformations that reduce dynamic power dissipation in low-power synthesis. In this case study, based on [19], we discuss how to develop a fast sequential simulator using AIGs.

5.1 Problem Formulation

The design is sequentially simulated for a fixed number of time-frames. A sequential simulator applies, at each time step, a set of values to the PIs. In the simplest case,

random PI patterns are generated to have a 0.5 probability of the node changing its value (fixed toggle rate). In other scenarios, the probability of an input transitions is given by the user, or produced by another tool. For example, if an input trace is known, it may be used for simulating the design. It is assumed that the initial state is known and initializes the sequential elements at the first time-frame. In subsequent time frames, the state derived at the previous iteration is used.

The runtime of sequential simulation can be reduced by minimizing the memory footprint. This is because most CPUs have a local cache ranging in size from 2Mb to 16Mb. If an application requires more memory than this, repeated cache misses cause the runtime to degrade. Therefore, the challenge is to design a simulator that uses a minimalistic data-structure without compromising the computation speed.

We found three orthogonal ways of reducing the memory requirements of the simulator, which in concert greatly improve its performance.

Compacting logic representation. Sequential designs are represented as AIGs. A typical AIG package uses 32 or more bytes to represent each AIG object (an internal AND node, an PI/PO, or a flop outputs/inputs). However, a minimalistic AIG package requires only 8 bytes per object. For an internal node, two integer fields, four bytes each, are used to store the fanin IDs. Other data structures may be temporarily allocated, for example, a hash-table for structural hashing may be used during AIG construction and deallocated before simulation begins.

Recycling simulation memory. When simulation is applied to a large sequential design, storing simulated values for all nodes in each timeframe requires a lot of memory. One way of avoid this, is to use the simulation information as soon as it is computed and to recycle the memory when it is not needed. For example, to estimate switching activity, we are only interested in counting the number of transitions seen at each node. For this, an integer counter can be used, thereby adding four bytes per object to the AIG package memory requirements, while the simulation information does not have to be stored.

Additionally, there is no need to allocate simulation memory for each object in the AIG. At any time during simulation, we only need to store simulation values for each combinational input/output and the nodes on the simulation frontier. These are all the nodes whose fanins are already simulated but at least one fanout is not yet simulated. For industrial designs, the number of internal nodes where simulation information should be stored is typically very small. For example, large industrial designs tend to have simulation frontier that is less than 1% of the total number of AIG nodes. The notion of a simulation frontier has also been useful to reduce memory requirements for the representation of priority cuts [28].

Bit-parallel simulation of two time-frames at the same time. A naïve approach to estimate the transition probability for each AIG node would be to store simulation patterns in two consecutive timeframes. Then, this information is compared (using bitwise XOR), and the number of ones in the bitwise representation is accumulated while simulating the timeframes. However, saving simulation information at each node for two consecutive timeframes leads to a large memory footprint. For example, an AIG with 1M objects requires 80Mb to store the simulation information for two timeframes, assuming 10 machine words (40 bytes) per object.

This increase in memory can be avoided by simultaneously simulating data belonging to two consecutive timeframes. In this case, comparison across the timeframes is made immediately, without memorizing previously computed results. This leads to duplicating the computation effort by simulating every pattern twice, one with the previous state value and the other with the current state value. However, the speedup due to not having to traverse the additional memory (causing excessive cash misses) outweighs the disadvantage of the re-computation.

5.2 Experimental Results

This section summarizes two experiments performed to evaluate the new simulator.

The first experiment, was designed to show that the new sequential simulator, called *SimSwitch,* has affordable runtimes for large designs. Four industrial designs ranging from 304K to 1.3M AIG nodes were simulated with different numbers of simulation patterns, ranging from 2,560 to 20,480. The input toggle rate was assumed to be 0.5. The results are shown in Table 1. Columns "AIG" and "FF" show the numbers of AIG nodes and registers. The runtimes for different amounts of input patterns are shown in the last columns. Note that the runtimes are quite affordable even for the design with 1.3M AIG nodes. In all four cases, the 2,560 patterns were sufficient for node switching activity rates to converge to a steady state.

Table 1. Runtime of *SimSwitch*

Design	AIG	FF	Runtime for inputs patterns (seconds)			
			2560	5120	10240	20480
C1	304K	1585	0.1	0.2	0.2	0.4
C2	362K	27514	2.7	2.9	4.1	6.6
C3	842K	58322	7.4	7.6	10.2	18.2
C4	1306K	87157	12.1	15.4	15.7	24.2

In the second experiment, we compare the runtime of S*imSwitch* vs. ACE-2.0 on 14 industry designs and 12 large academic benchmarks. The input toggle rate is assumed to be 0.5 for both tools. The number of input patterns is assumed to be 5,000 for both runs. All circuits are decomposed into AIG netlists before performing the switching estimation. The table of results can be found in [19]. The summary of results are as follows:

- For industry designs, *SimSwitch* is 149+ times faster than ACE-2.0.
- For academic benchmarks, SimSwitch is 85+ times faster than ACE-2.0.
- *SimSwitch* finished all testcases while ACE-2.0 times out on four industrial designs.

6 Optimization and Verification Flows

This section describes integrated sequences of transformations applied in ABC.

6.1 Integration of Synthesis

The optimization algorithms of ABC are integrated into a system called Magic [31] and interfaced with a design database developed to store realistic industrial designs. For instance, Magic handles multiple clock domains, flip-flops with complex controls, and special objects such as adder chains, RAMs, DSP modules, etc. Magic was developed to work with hierarchical designs whose sequential logic cones, when represented as a monolithic AIG, contain more than 1M nodes. The algorithms are described in the following publications:

Synthesis
- Scalable sequential synthesis [29] and retiming [34].
- Combinational synthesis using AIG rewriting [26].
- Combinational restructuring for delay optimization [30].

Mapping
- Mapping with structural choices [14].
- Mapping with global view and priority cuts [28].
- Mapping to optimize application-specific metrics [18][19].

Verification
- Fast sequential simulation [19]
- Improved combinational equivalence checking [27].
- Improved sequential equivalence checking [29][33].

The integration of components inside Magic is shown in Fig. 2. The design database is the central component interfacing the application packages. The design entry into Magic is performed through a file or via programmable APIs.

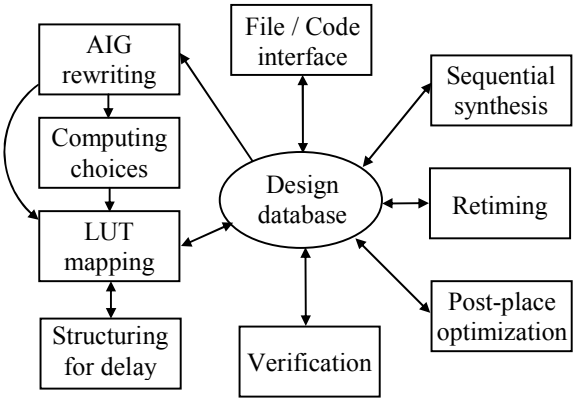


Fig. 2. Interaction of application packages in Magic

Shown on the right of Fig. 2, is sequential synthesis based on detecting, proving, and merging sequentially equivalent nodes. This transformation can be applied at the

beginning of the flow, before combinational synthesis and mapping. Another optional transform is retiming that reduces the total number of logic levels in the AIG or in the mapped network. Reducing the number of logic levels correlates with but does not always lead to an improvement in the clock frequency after place-and-route. The sequential transforms can be verified by sequential simulation and sequential equivalence checking.

Shown on the left of Fig. 2, is the combinational synthesis flow, which includes AIG rewriting, computing structural choices, and FPGA look-up-table (LUT) mapping. Computation of structural choices can be skipped if fast low-effort synthesis is desired. The result of mapping is returned to the design database or passed on to restructuring for delay optimization. After combinational synthesis, the design can be verified using combinational equivalence checking.

Finally, the box in the bottom right corner represents post-placement resynthesis, which includes incremental restructuring and retiming with wire-delay information.

6.2 Integration of Verification

Similar to IBM's tool SixthSense [2], the verification subsystem of ABC is an integrated set of applications, divided into several categories: miter simplifiers (i.e. sequential synthesis), bug-hunters (i.e. bounded model checking), and provers (i.e. interpolation). The high-level interface coded in Python orchestrates the applications and determines the resource limits used. An embedded Python interpreter allows for defining new procedures in addition to those included.

An AIG file is read in, and the objective is to prove each output unsatisfiable or find a counter-example. The top-level functions are *prove* and *prove_g_pos.* The former works for single-output properties, while the latter applies the former to each output of a multi-output miter, or to several outputs grouped together based on the group's support. The main flow is

$$pre\_simp \rightarrow quick\_verify \rightarrow abstract \rightarrow quick\_verify \rightarrow speculate \rightarrow final\_verify,$$

with each function passing the resulting AIG to the next function. At each stage, a set of resources is selected to spend on an algorithm. These resources are: total time, limit on the number of conflicts in SAT, maximum number of timeframes to unroll, maximum number of BDD nodes, etc. The allocation of resources is guided by the state of verification and the AIG parameters (the number of PIs, POs, FFs, AIG nodes, BMC depth reached, etc), which vary when the AIG is simplified and abstracted.

A global parameter *x_factor* can be used to increase the resources. If the problem is proved UNSAT by one of the application packages, the computation stops and the result is returned. . If the problem is found SAT and no abstraction has been done, the counter-example is returned.

The function *pre_simp* tries to reduce the AIG by applying several simplification algorithms:

- Phase abstraction, trying to identify clock-like periodic behaviors and deciding to unfold the design several frames depending on the clocks found and the amount of simplification this may allow [6].
- Trimming, which eliminates PIs that have no fanouts.

- Constraint extraction, which looks for implicit constraints inductively, uses these to simplify the design, and folds them back in with a structure such that if ever a constraint is not satisfied, the output is forced to be 0 from then on [12].
- Forward retiming and sequential FF correspondence, which finds correspondences between FFs and reduces the FF count, especially in SEC examples [29].
- Strong simplification script *simplify*, which iterates AIG rewriting, retiming for minimum FF (flip-flop) count, *k*-step sequential signal correspondence with *k* selected based on problem size. Also, the effort spent in signal correspondence can be adjusted by using a dedicated circuit-based SAT solver.

If *simplify* has already been applied to an AIG, then repeating it is usually fast, so the verification flow iterates it several times when other reductions have been done.

The function *quick_verify*, performed after each significant AIG reduction, is a low resource version of *final_verify*. These functions try to prove the problem by running interpolation or, if the problem seems small enough, by attempting BDD reachability.

The algorithm *abstract* is a combination of counter-example abstraction and proof-based abstraction implemented in a single SAT instance [17]. It returns an abstracted version of the AIG (a set of registers removed and replaced by PIs) and the frame count it was able to explore. To double check that a valid abstraction is derived, BMC (or, if the problem is small enough, BDD reachability) is applied to the resulting abstraction using additional resources. If a counter-example is found, *abstract* is restarted with additional resources from the frame where the counter-example was found.

The algorithm *speculate* applies speculative reduction [32][33]. This algorithm finds candidate sequential equivalences in the AIG, and creates a speculative reduced model, by transferring the fanouts of each equivalence class to a single representative, while creating new outputs, which become additional proof obligations. This model is refined as counter-examples are produced, finally arriving at a model that has no counterexamples up to some depth explored by BMC. Then, attempts are made to prove the outputs of the speculatively reduced model. If all outputs are successfully proved, the initial verification problem is solved. If at least one of the outputs failed, the candidate equivalences have to be filtered and speculative reduction repeated.

6.3 Example of Running the Verification Flow

Below is an example of a printout produced by ABC during verification of an industrial design. Comments follow the printout.

```
abc> Read_file  example1.aig
PIs = 532, POs = 1, FF = 2389, ANDs = 12049
abc> prove

Simplifying
Number of constraints found = 3
Forward retiming, quick_simp, scorr_comp, trm: PIs = 532, POs = 1, FF = 2342,
ANDs = 11054
Simplify:    PIs = 532, POs = 1, FF = 2335, ANDs = 10607
Phase abstraction:    PIs = 283, POs = 2, FF = 1460, ANDs = 8911
```

Abstracting
Initial abstraction: PIs = 1624, POs = 2, FF = 119, ANDs = 1716, max depth = 39
Testing with BMC
bmc3 -C 100000 -T 50 -F 78: No CEX found in 51 frames
Latches reduced from 1460 to 119
Simplify: PIs = 1624, POs = 2, FF = 119, ANDs = 1687, max depth = 51
Trimming: PIs = 158, POs = 2, FF = 119, ANDs = 734, max depth = 51
Simplify: PIs = 158, POs = 2, FF = 119, ANDs = 731, max depth = 51

Speculating
Initial speculation: PIs = 158, POs = 26, FF = 119, ANDs = 578, max depth = 51
Fast interpolation: reduced POs to 24
Testing with BMC
bmc3 -C 150000 -T 75: No CEX found in 1999 frames
PIs = 158, POs = 24, FF = 119, ANDs = 578, max depth = 1999
Simplify: PIs = 158, POs = 24, FF = 119, ANDs = 535, max depth = 1999
Trimming: PIs = 86, POs = 24, FF = 119, ANDs = 513, max depth = 1999

Verifying
Running reach -v -B 1000000 -F 10000 -T 75: BDD reachability aborted
RUNNING interpolation with 20000 conflicts, 50 sec, max 100 frames: 'UNSAT'

Elapsed time: 457.87 seconds, total: 458.52 seconds

NOTES:

1. The file *example1.aig* is first read in and its statistics are reported: 532 primary inputs, 1 primary output, 2389 flip-flops, and 12049 AIG nodes.
2. 3 implicit constraints were found, but they turned out to be only mildly useful in simplifying the problem.
3. Phase abstraction found a cycle of length 2 and this was useful for simplifying the problem to 1460 FF from 2335 FF. Note that the number of outputs increased to 2 because the problem was unrolled 2 time frames.
4. Abstraction was successful in reducing the FF count to 119. This was proved valid out to 39 time frames.
5. BMC verified that the abstraction produced is actually valid to 51 frames, which gives us good confidence that the abstraction is valid for all time.
6. Trimming reduced the inputs relevant to the abstraction from 1624 to 158 and *simplify* reduced the number of AIG nodes to 731.
7. Speculation produced a speculative reduced model (SRM) with 24 new outputs to be proved and low resource interpolation proved 2 of them. The SRM model is simpler and has only 578 AIG nodes. The SRM was tested with BMC and proved valid out to 1999 frames.
8. Subsequent trimming and simplification reduced the PIs to 86 and AIG size to 513.
9. The final verification step first tried BDD reachability allowing it 75 sec. and to grow to up to 1M BDD nodes. It could not converge with these resources so it was aborted. Then interpolation has returned UNSAT, and hence all 24 outputs are proved.

10. Although *quick_verify* was applied between simplification and abstraction, and between abstraction and speculation, it was not able to prove anything, so its output is not shown.
11. The total time was 457 seconds on a Lenovo X301 laptop with 1.4Gb Intel Core2 Duo CPU and 3Gb RAM.

7 Conclusions and Future Work

In this paper, we discussed the development of ABC and described its basic principles. Started five years ago, ABC continues to grow and gain momentum as a public-domain tool for logic synthesis and verification. New implementations, improvements, bug fixes, and performance tunings are added frequently. Even the core computations continue to improve through better implementation and exploiting the synergy between synthesis and verification. Possibly another 2-5x speedup can be obtained in these computations using the latest findings in the field. As always, a gain in runtime allows us to perform more iterations of synthesis with larger resource limits, resulting in stronger verification capabilities.

Future work will continue in the following directions:

- Improving core applications, such as AIG rewriting (by partitioning the problem and prioritizing rewriting moves) and technology mapping (by specializing the mapper to an architecture based on a given lookup-table size or a given programmable cell).
- Developing new applications (for example, a fast incremental circuit-based SAT solver or a back-end prover based on an OR-decomposition of the property cone, targetting properties not provable by known methods).
- Building industrial optimization/mapping/verification flows, such as Magic [31], targeting other implementation technologies (for example, the FPGA synthesis flow can be extended to work for standard cells).
- Disseminating the innovative principles of building efficient AIG/SAT/ simulation applications and the ways of exploiting the synergy of synthesis and verification.
- Customizing ABC for users in such domains as software synthesis, cryptography, computational biology, etc.

ABC is available for free from Berkeley Verification and Synthesis Research Center [3].

Acknowledgement

This work has been supported in part by SRC contracts 1361.001 and 1444.001, NSF grant CCF-0702668, and the industrial sponsors: Abound Logic, Actel, Altera, Atrenta, Calypto, IBM, Intel, Intrinsity, Magma, Mentor Graphics, Synopsys, Synplicity (Synopsys), Tabula, Verific, and Xilinx.

References

[1] Baumgartner, J., Kuehlmann, A.: Min-area retiming on flexible circuit structures. In: Proc. ICCAD '01, pp. 176–182 (2001)

[2] Baumgartner, J., Mony, H., Paruthi, V., Kanzelman, R., Janssen, G.: Scalable sequential equivalence checking across arbitrary design transformations. In: Proc. ICCD '06 (2006)

[3] Berkeley Verification and Synthesis Research Center (BVSRC), http://www.bvsrc.org

[4] Biere, A.: AIGER: A format for And-Inverter Graphs, http://fmv.jku.at/aiger/

[5] Bjesse, P., Boralv, A.: DAG-aware circuit compression for formal verification. In: Proc. ICCAD '04, pp. 42–49 (2004)

[6] Bjesse, P., Kukula, J.H.: Automatic generalized phase abstraction for formal verification. In: Proc. ICCAD '05, pp. 1076–1082 (2005)

[7] Brand, D.: Verification of large synthesized designs. In: Proc. ICCAD '93, pp. 534–537 (1993)

[8] Brayton, R.K., Rudell, R., Sangiovanni-Vincentelli, A.L., Wang, A.R.: MIS: A multiple-level logic optimization system. IEEE Trans. CAD 6(6), 1062–1081 (1987)

[9] Brayton, R.K., Hachtel, G.D., Sangiovanni-Vincentelli, A., Somenzi, F., Aziz, A., Cheng, S.-T., Edwards, S., Khatri, S., Kukimoto, Y., Pardo, A., Qadeer, S., Ranjan, R.K., Sarwary, S., Shiple, T.R., Swamy, G., Villa, T.: VIS: A system for verification and synthesis. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102. Springer, Heidelberg (1996)

[10] Brayton, R.: The synergy between logic synthesis and equivalence checking. In: Keynote at FMCAD'07 (2007), http://www.cs.utexas.edu/users/hunt/FMCAD/2007/presentations/fmcad07_brayton.ppt

[11] Bryant, R.E.: Graph based algorithms for Boolean function manipulation. IEEE TC 35(8), 677–691 (1986)

[12] Cabodi, G., Camurati, P., Garcia, L., Murciano, M., Nocco, S., Quer, S.: Speeding up model checking by exploiting explicit and hidden verification constraints. In: Proc. DATE '09, pp. 1686–1691 (2009)

[13] Chai, D., Jiang, J.-H., Jiang, Y., Li, Y., Mishchenko, A., Brayton, R.: MVSIS 2.0 programmer's manual. UC Berkeley (May 2003)

[14] Chatterjee, S., Mishchenko, A., Brayton, R., Wang, X., Kam, T.: Reducing structural bias in technology mapping. In: Proc. ICCAD '05, pp. 519–526 (2005), http://www.eecs.berkeley.edu/~alanmi/publications/2005/iccad05_map.pdf

[15] Coudert, O., Berthet, C., Madre, J.C.: Verification of sequential machines based on symbolic execution. In: Sifakis, J. (ed.) CAV 1989. LNCS, vol. 407. Springer, Heidelberg (1990)

[16] Darringer, A., Joyner Jr., W.H., Berman, C.L., Trevillyan, L.: Logic synthesis through local transformations. IBM J. of Research and Development 25(4), 272–280 (1981)

[17] Een, N., Mishchenko, A., Amla, N.: A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. In: Proc. IWLS'10 (2010)

[18] Jang, S., Chan, B., Chung, K., Mishchenko, A.: WireMap: FGPA technology mapping for improved routability. In: Proc. FPGA '08, pp. 47–55 (2008)

[19] Jang, S., Chung, K., Mishchenko, A., Brayton, R.: A power optimization toolbox for logic synthesis and mapping. In: Proc. IWLS '09, pp. 1–8 (2009)

[20] Jiang, J.-H.R., Brayton, R.: Retiming and resynthesis: A complexity perspective. IEEE Trans. CAD 25(12), 2674–2686 (2006), http://www.eecs.berkeley.edu/~brayton/publications/2006/tcad06_r&r.pdf
[21] Jiang, J.-H.R., Hung, W.-L.: Inductive equivalence checking under retiming and resynthesis. In: Proc. ICCAD'07, pp. 326–333 (2007)
[22] Kuehlmann, A., Paruthi, V., Krohm, F., Ganai, M.K.: Robust Boolean reasoning for equivalence checking and functional property verification. IEEE Trans. CAD 21(12), 1377–1394 (2002)
[23] Leiserson, C.E., Saxe, J.B.: Retiming synchronous circuitry. Algorithmica 6, 5–35 (1991)
[24] Lamoureux, J., Wilton, S.J.E.: Activity estimation for Field-Programmable Gate Arrays. In: Proc. Intl Conf. Field-Prog. Logic and Applications (FPL), pp. 87–94 (2006)
[25] Mishchenko, A., Zhang, J.S., Sinha, S., Burch, J.R., Brayton, R., Chrzanowska-Jeske, M.: Using simulation and satisfiability to compute flexibilities in Boolean networks. IEEE Trans. CAD 25(5), 743–755 (2006)
[26] Mishchenko, A., Chatterjee, S., Brayton, R.: DAG-aware AIG rewriting: A fresh look at combinational logic synthesis. In: Proc. DAC '06, pp. 532–536 (2006), http://www.eecs.berkeley.edu/~alanmi/publications/2006/dac06_rwr.pdf
[27] Mishchenko, A., Chatterjee, S., Brayton, R., Een, N.: Improvements to combinational equivalence checking. In: Proc. ICCAD '06, pp. 836–843 (2006)
[28] Mishchenko, A., Cho, S., Chatterjee, S., Brayton, R.: Combinational and sequential mapping with priority cuts. In: Proc. ICCAD '07, pp. 354–361 (2007)
[29] Mishchenko, A., Case, M.L., Brayton, R.K., Jang, S.: Scalable and scalably-verifiable sequential synthesis. In: Proc. ICCAD'08, pp. 234–241 (2008)
[30] Mishchenko, A., Brayton, R., Jang, S.: Global delay optimization using structural choices. In: Proc. FPGA'10, pp. 181–184 (2010)
[31] Mishchenko, A., Een, N., Brayton, R.K., Jang, S., Ciesielski, M., Daniel, T.: Magic: An industrial-strength logic optimization, technology mapping, and formal verification tool. In: IWLS'10 (2010)
[32] Mony, H., Baumgartner, J., Paruthi, V., Kanzelman, R.: Exploiting suspected redundancy without proving it. In: Proc. DAC'05, pp. 463–466 (2005)
[33] Mony, H., Baumgartner, J., Mishchenko, A., Brayton, R.: Speculative reduction-based scalable redundancy identification. In: Proc. DATE'09, pp. 1674–1679 (2009)
[34] Ray, S., Mishchenko, A., Brayton, R.K., Jang, S., Daniel, T.: Minimum-perturbation retiming for delay optimization. In: Proc. IWLS'10 (2010)
[35] Sentovich, E.M., Singh, K.J., Lavagno, L., Moon, C., Murgai, R., Saldanha, A., Savoj, H., Stephan, P.R., Brayton, R.K., Sangiovanni-vincentelli, A.: SIS: A system for sequential circuit synthesis. Technical Report, UCB/ERI, M92/41, ERL, Dept. of EECS, UC Berkeley (1992)
[36] Somenzi, F.: BDD package. CUDD v. 2.3.1, http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html
[37] Yang, C., Ciesielski, M., Singhal, V.: BDS: a BDD-based logic optimization system. In: Proc. DAC'00, pp. 92–97 (2000), http://www.ecs.umass.edu/ece/labs/vlsicad/bds/bds.html

There's Plenty of Room at the Bottom: Analyzing and Verifying Machine Code*
(Invited Tutorial)

Thomas Reps[1,2,**], Junghee Lim[1], Aditya Thakur[1], Gogul Balakrishnan[3], and Akash Lal[4]

[1] University of Wisconsin; Madison, WI, USA
[2] GrammaTech, Inc.; Ithaca, NY, USA
[3] NEC Laboratories America, Inc.; Princeton, NJ, USA
[4] Microsoft Research India; Bangalore, India

Abstract. This paper discusses the obstacles that stand in the way of doing a good job of machine-code analysis. Compared with analysis of source code, the challenge is to drop all assumptions about having certain kinds of information available (variables, control-flow graph, call-graph, etc.) and also to address new kinds of behaviors (arithmetic on addresses, jumps to "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream, self-modifying code, etc.).

The paper describes some of the challenges that arise when analyzing machine code, and what can be done about them. It also provides a rationale for some of the design decisions made in the machine-code-analysis tools that we have built over the past few years.

1 Introduction

This paper is intended to complement the papers that we have written over the past few years on verifying safety properties of stripped executables. Elsewhere (e.g., [9] and [3, §1]) we have argued at length the benefits of analyzing machine code rather than source code. In brief,

- Machine code is an artifact that is closer to what actually executes on the machine; models derived from machine code can be more accurate than models derived from source code (particularly because compilation, optimization, and link-time transformation can change how the code behaves).
- When source code is compiled, the compiler and optimizer make certain choices that eliminate some possible behaviors—hence there is sometimes the opportunity to obtain more precise answers from machine-code analysis than from source-code analysis.

* Supported, in part, by NSF under grants CCF-{0540955, 0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 09-1-0776}, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.

** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology discussed in this publication.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 41–56, 2010.

Rather than rehashing those arguments here, we take them as givens, and focus instead on the obstacles standing in the way of doing a good job of machine-code analysis. The paper explains some of the challenges that arise when analyzing machine code, and what can be done about them. It thereby provides a rationale for some of the design decisions made in the machine-code-analysis tools that we have built over the past few years, in particular CodeSurfer/x86 [8,3], DDA/x86 [7], and MCVETO [27]. Those three tools represent several firsts:

- CodeSurfer/x86 is the first program-slicing tool for machine code that is able to track the flow of values through memory, and thus help with understanding dependences transmitted via memory loads and stores.
- DDA/x86 is the first automatic program-verification tool that is able to check whether a stripped executable—such as a device driver—conforms to an API-usage rule (specified as a finite-state machine).
- MCVETO is the first automatic program-verification tool capable of verifying (or detecting flaws in) self-modifying code.

As with any verification tool, each of these tools comes with a few caveats about the class of programs to which it can be applied, which are due to certain design decisions concerning the analysis techniques used.

The remainder of the paper is organized as follows: §2 describes some of the challenges presented by machine-code analysis and verification, as well as different aspects of the design space for analysis and verification tools. §3 discusses one point in the design space: when the goal is to account only for behaviors expected from a standard compilation model, but report evidence of possible deviations from such behaviors. §4 discusses another point in the design space: when the goal is to verify machine code, including accounting for deviant behaviors. §5 discusses how we have created a way to build "Yacc-like" tool generators for machine-code analysis and verification tools (i.e., from a semantic specification of a language L, we are able to create automatically an instantiation of a given tool for L). §6 concerns related work. (Portions of the paper are based on material published elsewhere, e.g., [7,3,21,27].)

2 The Design Space for Machine-Code Analysis

Machine-code-analysis problems come in at least three varieties: (i) in addition to the executable, the program's source code is also available; (ii) the source code is unavailable, but the executable includes symbol-table/debugging information ("unstripped executables"); (iii) the executable has no symbol-table/debugging information ("stripped executables"). The appropriate variant to work with depends on the intended application. Some analysis techniques apply to multiple variants, but other techniques are severely hampered when symbol-table/debugging information is absent. In our work, we have primarily been concerned with the analysis of stripped executables, both because it is the most challenging situation, and because it is what is needed in the common

situation where one needs to install a device driver or commercial off-the-shelf application delivered as stripped machine code. If an individual or company wishes to vet such programs for bugs, security vulnerabilities, or malicious code (e.g., back doors, time bombs, or logic bombs) analysis tools for stripped executables are required.

Compared with source-code analysis, analysis of stripped executables presents several problems. In particular, standard approaches to source-code analysis assume that certain information is available—or at least obtainable by separate analysis phases with limited interactions between phases, e.g.,

- a control-flow graph (CFG), or interprocedural CFG (ICFG)
- a call-graph
- a set of variables, split into disjoint sets of local and global variables
- a set of non-overlapping procedures
- type information
- points-to information or alias information

The availability of such information permits the use of techniques that can greatly aid the analysis task. For instance, when one can assume that (i) the program's variables can be split into (a) global variables and (b) local variables that are encapsulated in a conceptually protected environment, and (ii) a procedure's return address is never corrupted, analyzers often tabulate and reuse explicit summaries that characterize a procedure's behavior.

Source-code-analysis tools sometimes also use questionable techniques, such as interpreting operations in integer arithmetic, rather than bit-vector arithmetic. They also usually make assumptions about the semantics that are not true at the machine-code level—for instance, they usually assume that the area of memory beyond the top-of-stack is not part of the execution state at all (i.e., they adopt the fiction that such memory does not exist).

In general, analysis of stripped executables presents many challenges and difficulties, including

absence of information about variables: In stripped executables, no information is provided about the program's global and local variables.

a semantics based on a flat memory model: With machine code, there is no notion of separate "protected" storage areas for the local variables of different procedure invocations, nor any notion of protected fields of an activation record. For instance, a procedure's return address is stored on the stack; an analyzer must prove that it is not corrupted, or discover what new values it could have.

absence of type information: In particular, int-valued and address-valued quantities are indistinguishable at runtime.

arithmetic on addresses is used extensively: Moreover, numeric and address-dereference operations are inextricably intertwined, even during simple operations. For instance, consider the load of a local variable `v`, located at offset

```
void foo() {
  int arr[2], n;
  void (*addr_bar)() = bar;
  if(MakeChoice() == 7) n = 4; // (*)
  else n = 2;
  for(int i = 0; i < n; i++)
    arr[i] = (int)addr_bar; // (**)
  return; // can return to the entry of bar
}

void bar() {
  ERR: return;
}

int main() {
  foo();
  return 0;
}
```

Fig. 1. A program that, on some executions, can modify the return address of `foo` so that `foo` returns to the beginning of `bar`, thereby reaching `ERR`. (`MakeChoice` is a primitive that returns a random 32-bit number).

`-12` in the current activation record, into register `eax`: `mov eax,[ebp-12]`.[1] This instruction involves a *numeric* operation (`ebp-12`) to calculate an address whose value is then *dereferenced* (`[ebp-12]`) to fetch the value of `v`, after which the value is placed in `eax`.

instruction aliasing: Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [22].

self-modifying code: With self-modifying code there is no fixed association between an address and the instruction at that address.

Because certain kinds of information ordinarily available during source-code analysis (variables, control-flow graph, call-graph, etc.) are not available when analyzing machine code, some standard techniques are precluded. For instance, source-code analysis tools often use separate phases of (i) points-to/alias analysis (analysis of addresses) and (ii) analysis of arithmetic operations. Because numeric and address-dereference operations are inextricably intertwined, as discussed above, only very imprecise information would result with the same organization of analysis phases.

[1] For readers who need a brief introduction to the 32-bit Intel x86 instruction set (also called IA32), it has six 32-bit general-purpose registers (`eax`, `ebx`, `ecx`, `edx`, `esi`, and `edi`), plus two additional registers: `ebp`, the frame pointer, and `esp`, the stack pointer. By convention, register `eax` is used to pass back the return value from a function call. In Intel assembly syntax, the movement of data is from right to left (e.g., `mov eax,ecx` sets the value of `eax` to the value of `ecx`). Arithmetic and logical instructions are primarily two-address instructions (e.g., `add eax,ecx` performs `eax := eax + ecx`). An operand in square brackets denotes a dereference (e.g., if `v` is a local variable stored at offset -12 off the frame pointer, `mov [ebp-12],ecx` performs `v := ecx`). Branching is carried out according to the values of condition codes ("flags") set by an earlier instruction. For instance, to branch to `L1` when `eax` and `ebx` are equal, one performs `cmp eax,ebx`, which sets `ZF` (the zero flag) to 1 iff `eax` − `ebx` = 0. At a subsequent jump instruction `jz L1`, control is transferred to `L1` if `ZF` = 1; otherwise, control falls through.

Fig. 1 is an example that will be used to illustrate two points in the design space of machine-code-analysis tools with respect to the question of corruption of a procedure's return address. When the program shown in Fig. 1 is compiled with Visual Studio 2005, the return address is located two 4-byte words beyond `arr`—in essence, at `arr[3]`. When `MakeChoice` returns 7 at line (*), `n` is set to 4, and thus in the loop `arr[3]` is set to the starting address of procedure `bar`. Consequently, the execution of `foo` can modify `foo`'s return address so that `foo` returns to the beginning of `bar`.

In general, tools that represent different points in the design space have different answers to the question

What properties are checked, and what is expected of the analyzer after the first anomalous action is detected?

First, consider the actions of a typical source-code analyzer, which would propagate abstract states through an interprocedural control-flow graph (ICFG). The call on `foo` in `main` causes it to begin analyzing `foo`. Once it is finished analyzing `foo`, it would follow the "return-edge" in the ICFG back to the point in `main` after the call on `foo`. However, a typical source-code analyzer does not represent the return address explicitly in the abstract state and relies on an unsound assumption that the return address cannot be modified. The analyzer would never analyze the path from `main` to `foo` to `bar`, and would thus miss one of the program's possible behaviors. The analyzer might report an array-out-of-bounds error at line (**).

As explained in more detail in §3, in CodeSurfer/x86 and DDA/x86, we were able to make our analysis problems resemble standard source-code analysis problems, to a considerable degree. One difference is that in CodeSurfer/x86 and DDA/x86 the return address is represented explicitly in the abstract state. At a return, the current (abstract) value of the return address is checked against the expected value. If the return address is not guaranteed to have the expected value, a report is issued that the return

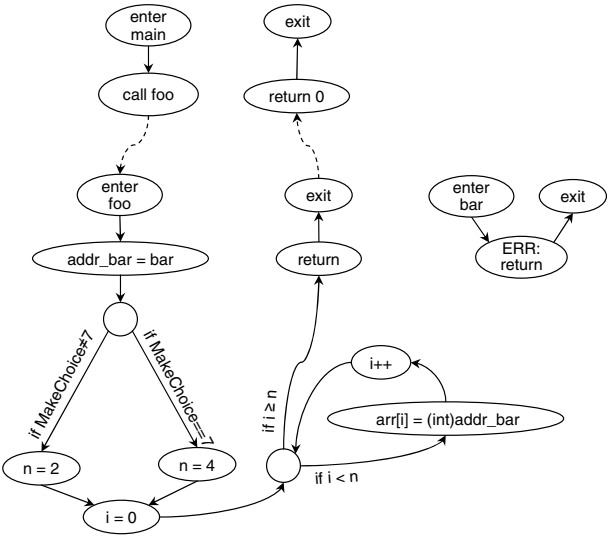


Fig. 2. Conventional ICFG for the program shown in Fig. 1. Note that the CFG for `bar` is disconnected from the rest of the ICFG.

address may have been modified. However, for reasons explained in §3, the analyses used in CodeSurfer/x86 and DDA/x86 proceed according to the original return address—i.e., by returning from `foo` to `main`. Similar to source-code analyzers, they would not analyze the path from `main` to `foo` to `bar`. Although they miss one of the program's possible behaviors, they report that there is possibly an anomalous overwrite of the return address.

In contrast, MCVETO uses some techniques that permit it not only to detect the presence of "deviant behaviors", but also to explore them as well. The state-space-exploration method used in MCVETO discovers that the execution of `foo` can modify `foo`'s return address. It uses the modified return address to discover that `foo` actually returns to the beginning of `bar`, and correctly reports that `ERR` is reachable.

3 Accounting for Behaviors Expected from a Standard Compilation Model

As illustrated in §2, CodeSurfer/x86[2] only follows behaviors expected from a standard compilation model. It is prepared to detect and report deviations from such behaviors, but not prepared to explore the consequences of deviant behavior. By a "standard compilation model", we mean that the executable has procedures, activation records (ARs), a global data region, and a free-storage pool; might use virtual functions and DLLs; maintains a runtime stack; each global variable resides at a fixed offset in memory; each local variable of a procedure f resides at a fixed offset in the ARs for f; actual parameters of f are pushed onto the stack by the caller so that the corresponding formal parameters reside at fixed offsets in the ARs for f; the program's instructions occupy a fixed area of memory, and are not self-modifying.

During the analyses performed by CodeSurfer/x86, these aspects of the program are checked. When violations are detected, an error report is issued, and the analysis proceeds. In doing so, however, we generally chose to have CodeSurfer/x86's analysis algorithms only explore behaviors that stay within those of the desired execution model. For instance, as discussed in §2, if the analysis discovers that the return address might be modified within a procedure, CodeSurfer/x86 reports the potential violation, but proceeds without modifying the control flow of the program. In the case of self-modifying code, either a write into the code will be reported or a jump or call to data will be reported.

If the executable conforms to the standard compilation model, CodeSurfer/x86 returns valid analysis results for it; if the executable does not conform to the model, then one or more violations will be discovered, and corresponding error reports will be issued; if the (human) analyst can determine that the error report is indeed a false positive, then the analysis results are valid. The advantages of

[2] Henceforth, we will not refer to DDA/x86 explicitly. Essentially all of the observations made about CodeSurfer/x86 apply to DDA/x86 as well.

this approach are three-fold: (i) it provides the ability to analyze some aspects of programs that may deviate from the desired execution model; (ii) it generates reports of possible deviations from the desired execution model; (iii) it does not force the analyzer to explore all of the consequences of each (apparent) deviation, which may be a false positive due to loss of precision that occurs during static analysis. If a deviation is possible, then at least one report will be a true positive: each possible *first* violation will be reported.

Memory Model. Although in the concrete semantics of x86 machine code the activation records for procedures, the heap, and the memory area for global data are all part of one address space, for the purposes of analysis, CodeSurfer/x86 adopts an approach that is similar to that used in source-code analyzers: the address space is treated as being separated into a set of disjoint areas, which are referred to as *memory-regions*. Each memory-region represents a group of locations that have similar runtime properties; in particular, the runtime locations that belong to the ARs of a given procedure belong to one memory-region. Each (abstract) byte in a memory-region represents a set of concrete memory locations. For a given program, there are three kinds of regions: (1) the global-region, for memory locations that hold initialized and uninitialized global data, (2) AR-regions, each of which contains the locations of the ARs of a particular procedure, and (3) malloc-regions, each of which contains the locations allocated at a particular malloc site [5].

All data objects, whether local, global, or in the heap, are treated in a fashion similar to the way compilers arrange to access variables in local ARs, namely, via an offset: an abstract address in a memory-region is represented by a pair: (memory-region, offset). For an n-bit architecture, the size of each memory-region in the abstract memory model is 2^n. For each region, the range of offsets within the memory-region is $[-2^{n-1}, 2^{n-1} - 1]$. Offset 0 in an AR-region represents all concrete starting addresses of the ARs that the AR-region represents. Offset 0 in a malloc-region represents all concrete starting addresses of the heap blocks that the malloc-region represents. Offset 0 of the global-region represents the concrete address 0. Nothing is assumed about the relative positions of memory-regions.

Analysis Algorithms. To a substantial degree, the analysis algorithms used in CodeSurfer/x86 closely resemble standard source-code analyses, although considerable work was necessary to map ideas from source-code analysis over to machine-code analysis. One of the main themes of the work on CodeSurfer/x86 was how an analyzer can bootstrap itself from preliminary intermediate representations (IRs) that record fairly basic information about the code of a stripped executable to IRs on which it is possible to run analyses that resemble standard source-code analyses. (See [3, §2.2,§4, and §5].)

The analyses used in CodeSurfer/x86 address the following problem:

Given a (possibly stripped) executable E, identify the procedures, data objects, types, and libraries that it uses, and,

- *for each instruction I in E and its libraries,*
- *for each interprocedural calling context of I, and*
- *for each machine register and variable V in scope at I,*

statically compute an accurate over-approximation to the set of values that V may contain when I executes.

The work presented in our 2004 paper [4] provided a way to apply the tools of abstract interpretation [12] to the problem of analyzing stripped executables (using the memory model sketched above) to statically compute an over-approximation at each program point to the set of values that a register or memory location could contain. We followed that work up with other techniques to complement and enhance the approach [26,19,25,5,6,2,7]. That body of work resulted in a method to recover a good approximation to an executable's variables and dynamically allocated memory objects, and to track the flow of values through them.

Caveats. Some of the limitations of CodeSurfer/x86 are due to the memory model that it uses. For instance, the memory-region-based memory model interferes with the ability to interpret masking operations applied to stack addresses. Rather than having *addr* & `MASK`, one has *(AR, offset)* & `MASK`, which generally results in $\top$ (i.e., any possible address) because nothing is known about the possible addresses of the base of *AR*, and hence nothing is known about the set of bit patterns that *(AR, offset)* represents. (Such masking operations are sometimes introduced by `gcc` to enforce stack alignment.)

4 Verification in the Presence of Deviant Behaviors

MCVETO has pioneered some techniques that permit it to verify safety properties of machine code, even if the program deviates from the behaviors expected from a standard compilation model. Because the goal is to account for deviant behaviors, the situation is more challenging than the one discussed in §3. For instance, in the case of self-modifying code, standard structures such as the ICFG and the call-graph are not even well-defined. That is, as discussed in §2, standard ways of interpreting the ICFG during analysis are not sound. One must look to other abstractions of the program's state space to accommodate such situations.

Our MCVETO tool addresses these issues by generalizing the source-code-analysis technique of directed proof generation (DPG) [16]. Given a program P and a particular control location *target* in P, DPG returns either an input for which execution leads to *target* or a proof that *target* is unreachable (or DPG does not terminate). DPG makes use of two approximations of P's state space:

- A set T of concrete traces, obtained by running P with specific inputs. T *under*approximates P's state space.
- A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G *over*approximates P's state space.

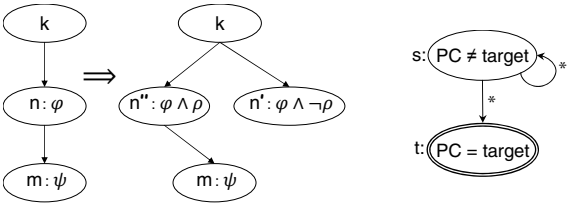


Fig. 3. (a) The general refinement step used in DPG. Refinement predicate ρ ensures that no execution can follow $n' \rightarrow m$. (b) The initial abstract graph used in McVeto. (* is a wild-card symbol that matches all instructions).

The two abstractions are played off one another, using the basic step from *directed test generation* [14] to determine whether it is possible to drive execution down a new path to *target*:

- If G has no path from *start* to *target*, then DPG has proven that *target* is unreachable, and G serves as the proof.
- If G has a path from *start* to *target* with a feasible prefix that has not been explored before, DPG initiates a concrete execution to attempt to reach *target*. Such a step augments the underapproximation T.
- If G has a path from *start* to *target* but the path has an infeasible prefix, DPG refines the overapproximation by performing the node-splitting operation shown in Fig. 3(a).

DPG is attractive for addressing the problem that we face, for two reasons. First, it is able to account for a program's deviant behaviors during the process of building up the underapproximation of the program's state space. Second, as we discuss below, the overapproximation of the program's state space can be constructed without relying on an ICFG or call-graph being available.

What Must be Handled Differently in Machine-Code DPG? The abstract graph used during DPG is an overapproximation of the program's state space. The versions of DPG used in SYNERGY [16], DASH [10], and SMASH [15] all start with an ICFG, which, when working with stripped machine code, is not only unavailable initially but may not even be well-defined. Nevertheless, for machine code, one can still create an over-approximation of the state space, as long as one makes a few adjustments to the basic elements of DPG.

1. The system needs to treat the value of the program counter (PC) as data so that predicates can refer to the value of the PC.
2. The system needs to learn the over-approximation starting with a cruder over-approximation than an ICFG. In particular, MCVETO starts from the initial abstraction shown in Fig. 3(b), which only has two abstract states, defined by the predicates "PC = *target*" and "PC $\neq$ *target*". The abstraction is gradually refined as more of the program is exercised.

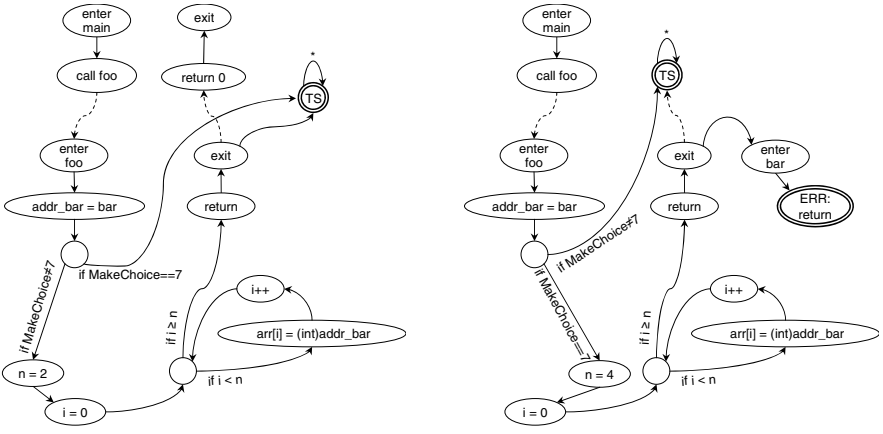


Fig. 4. Automata created by generalizing two execution traces. Each automaton contains an accepting state, called *TS* (for "target surrogate"). *TS* is accepting because it may represent *target*, as well as all non-*target* locations not visited by the trace.

3. To handle self-modifying code, a predicate that labels an abstract state may hold a constraint that specifies what instruction is decoded from memory, starting at the address held by the PC.
4. In addition to refinements of the abstract graph performed by the step shown in Fig. 3(a), the abstract graph is also refined each time a concrete execution fails to reach *target*. These refinements are inspired, in part, by the *trace-refinement* technique of Heizmann et al. [17]. The abstract graph is considered to be an automaton (e.g., s is a non-final state in Fig. 3(b), whereas target t is a final state). A concrete execution trace τ that reaches *target* is *minimal* if no proper prefix of τ reaches *target*. Each concrete execution trace that fails to reach *target* is *generalized* to create an automaton (or "folded trace") that accepts an overapproximation of the set of minimal concrete execution traces that reach *target*. The automaton is intersected with the current abstract graph to create the next version of the abstract graph.

The approach adopted by MCVETO has a number of advantages. First, it allows MCVETO to build a sound overapproximation of the program's state space on-the-fly, performing disassembly during state-space exploration, but never on more than one instruction at a time and without relying on a static split between code vs. data. In particular, MCVETO does not have to be prepared to disassemble *collections* of nested branches, loops, procedures, or the whole program all at once, which is what can confuse conventional disassemblers [22]. Second, because the abstraction of the program's state space is built entirely on-the-fly, it allows MCVETO to analyze programs with instruction aliasing. Third, it permits MCVETO to be able to verify (or detect flaws in) self-modifying code. With self-modifying code there is no fixed association between an address and the instruction at that address. However, by labeling each abstract state with a predicate

that refers to the address held by the PC, as well as a predicate that specifies what instruction is decoded from memory, starting at the address held by the PC, the abstract graph can capture relationships on an address, the instruction at that address, and the states that can arise for that ⟨address, instruction⟩ combination. A sound overapproximation of the program's state space is created automatically by MCVETO's two mechanisms for refining the abstract graph. Fourth, trace generalization allows eliminating *families* of infeasible traces. Compared to prior techniques that also have this ability [11,17], the technique involves no calls on an SMT solver, and avoids the potentially expensive step of automaton complementation (see [27, §3.1]).

Returning to the example from Figs. 1 and 2 of a procedure that can corrupt its return address, Fig. 4(a) shows the automaton obtained via trace generalization of the execution trace most likely to be performed during the initial execution. The directed-test-generation step then forces execution down the branch for `MakeChoice()==7`. In that execution, `foo` returns to the beginning of `bar`, from which `ERR` is reachable. (Fig. 4(b) shows the automaton that would be obtained via trace generalization from the second execution trace.)

```
int foo(int a) { return a; }

int main() {
  int n = 1; (*)
  __asm {
      mov eax, 0;
  L1: mov edx, 0xd0ff046a; // (**)
      add n, eax; // (***)
      cmp eax, 4;
      jz L2;
      mov eax, foo;
      lea ebx, L1+1;
      jmp ebx;
  L2: }
  if(n == 1)
      ERR:; // Unreachable
  return 0;
}
```

Fig. 5. A program that illustrates instruction aliasing. At line (**), when the instruction is read at the second byte, it becomes `L1+1: push 4; call eax`.

Fig. 5 shows a program that makes use of instruction aliasing. At line (**), when the instruction is read at the second byte (i.e., starting at `L1+1`), it becomes `L1+1: push 4; call eax`. `ERR` is unreachable because when the branch condition `if(n==1)` is evaluated, `n` always has the value 5: 1 from the initialization in line (*) plus 4 from the value of `eax` added in line (***), which is the return value from the hidden call to `foo` at line (**).

MCVETO builds an abstract graph based on the path

```
n=1; mov eax,0; L1: mov edx,0xd0ff046a; add n,eax; cmp eax,4; jz L2; mov
eax,foo; lea ebx,L1+1; jmp ebx; L1+1: push 4; call eax; return a; add
n,eax; cmp eax,4; jz L2; L2:; if(n==1); return 0
```

It then does a series of refinements of the abstract graph that culminate in a version in which there is no path from the beginning of the graph to `ERR`.

Discovering Candidate Invariants. To improve convergence, we introduced *speculative trace refinement*, which enhances the methods that MCVETO uses to refine the abstract graph. Speculative trace refinement was motivated by the

observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant. It involves first discovering invariants that hold for nodes of folded traces; the invariants are then incorporated into the abstract graph via automaton intersection. The basic idea is to apply dataflow analysis to a graph obtained from a folded trace to obtain invariants for its states. In the broader context of the full program, these are only *candidate* invariants. They are introduced into the abstract graph in the hope that they are also invariants of the full program. The recovery of invariants is similar in spirit to the computation of invariants from traces in Daikon [13], but in MCVETO they are computed *ex post facto* by dataflow analysis on a folded trace. Although the technique causes the abstract graph to be refined speculatively, the abstract graph is a sound overapproximation of the program's state space at all times.

We take this technique one step further for cases when proxies for program variables are needed in an analysis (e.g., affine-relation analysis [23]). Because no information is available about a program's global and local variables in stripped executables, we perform *aggregate-structure identification* [24] on a concrete trace to obtain a set of inferred memory variables. Because an analysis may not account for the full effects of indirect memory references on the inferred variables, to incorporate a discovered candidate invariant φ for node n into a folded trace safely, we split n on φ and $\neg\varphi$.

Caveats. MCVETO actually uses nested-word automata [1] rather than finite-state automata to represent the abstract graph and the folded traces that represent generalizations of execution traces. MCVETO makes the assumption that each `call` instruction represents a procedure call, and each `ret` instruction represents a return from a procedure call. This decision was motivated by the desire to have a DPG-based algorithm for verifying machine code that took advantage of the fact that most programs are well-behaved in most execution contexts. The consequence of this decision is that because MCVETO has some expectations on the behaviors of the program, for it to prove that *target* is unreachable it must also prove that the program cannot deviate from the set of expected behaviors (see [27, §3.5]). If a deviant behavior is discovered, it is reported and MCVETO terminates its search.

5 Automatic Tool Generation

Although CodeSurfer/x86 was based on analysis methods that are, in principle, language-independent, the original implementation was tied to the x86 instruction set. That situation is fairly typical of much work on program analysis: although the techniques described in the literature are, in principle, language-independent, implementations are often tied to one specific language. Retargeting them to another language can be an expensive and error-prone process. For machine-code analyses, having a language-dependent implementation is even worse than for source-code analyses because of the size and complexity of instruction sets. Because of instruction-set evolution over time (and the desire to have backward compatibility as word size increased from 8 bits to 64 bits), instruction

sets such as the x86 instruction set have several hundred kinds of instructions. Some instruction sets also have special features not found in other instruction sets. To address the problem of supporting multiple instruction sets, another aspect of our work on machine-code analysis has been to develop a meta-tool, or tool-generator, called TSL [21] (for **T**ransformer **S**pecification **L**anguage), to help in the creation of tools for analyzing machine code.

A tool generator (or tool-component generator) such as YACC [18] takes a declarative description of some desired behavior and automatically generates an implementation of a component that behaves in the desired way. Often the generated component consists of generated tables and code, plus some unchanging *driver* code that is used in each generated tool component. The advantage of a tool generator is that it creates correct-by-construction implementations.

For machine-code analysis, the desired components each consist of a suitable abstract interpretation of the instruction set, together with some kind of analysis driver (a solver for finding the fixed-point of a set of dataflow equations, a symbolic evaluator for performing symbolic execution, etc.). TSL is a system that takes a description of the concrete semantics of an instruction set, a description of an abstract interpretation, and creates an implementation of an abstract interpreter for the given instruction set.

$$\text{TSL} : \text{concrete semantics} \times \text{abstract domain} \rightarrow \text{abstract semantics}.$$

In that sense, TSL is a tool generator that, for a fixed instruction-set semantics, automatically creates different abstract interpreters for the instruction set.

An instruction set's concrete semantics is specified in TSL's input language, which is a strongly typed, first-order functional language with a datatype-definition mechanism for defining recursive datatypes, plus deconstruction by means of pattern matching. Writing a TSL specification for an instruction set is similar to writing an interpreter in first-order ML. For instance, the specification of an instruction set's concrete semantics is written as a TSL function

```
state interpInstr(instruction I, state S) {...};
```

where instruction and state are user-defined datatypes that represent the instructions and the semantic states, respectively.

TSL's input language provides a fixed set of base-types; a fixed set of arithmetic, bitwise, relational, and logical operators; and a facility for defining map-types. The meanings of the input-language constructs can be redefined by supplying alternative interpretations of them. When semantic reinterpretation is performed in this way—namely, on the operations of the input-language—it is independent of any given instruction set. Consequently, once a reinterpretation has been defined that reinterprets TSL in a manner appropriate for some state-space-exploration method, the same reinterpretation can be applied to each instruction set whose semantics has been specified in TSL.

The reinterpretation mechanism allows TSL to be used to implement *tool-component generators* and *tool generators*. Each implementation of an analysis component's driver (e.g., fixed-point-finding solver, symbolic executor) serves as the unchanging driver for use in different instantiations of the analysis compo-

nent for different instruction sets. The TSL language becomes the specification language for retargeting that analysis component for different instruction sets:

analyzer generator = abstract-semantics generator + analysis driver.

For tools like CodeSurfer/x86 and MCVETO, which incorporate multiple analysis components, we thereby obtain YACC-like tool generators for such tools:

concrete semantics of L → Tool/L.

Moreover, because all analysis components are generated from a single specification of the instruction set's concrete semantics, the generated implementations of the analysis components are guaranteed to be mutually consistent (and also to be consistent with an instruction-set emulator that is generated from the same specification of the concrete semantics).

As an example of the kind of leverage that TSL provides, the most recent incarnation of CodeSurfer/x86—a revised version whose analysis components are implemented via TSL—uses eight separate reinterpretations generated from the TSL specification of the x86 instruction set. The x86 version of MCVETO uses three additional reinterpretations [20] generated from the same TSL specification.

Discussion. MCVETO does not model all aspects of a machine-code program. For instance, it does not model timing-related behavior, the hardware caches, the Interrupt Descriptor Table (necessary for modeling interrupt-handler dispatch), etc. However, the use of TSL allows additional aspects to be added to the concrete operational semantics, independently from MCVETO's DPG algorithms. For example, although our current TSL description of the x86 instruction set does not model the Interrupt Descriptor Table, that is only a shortcoming of the current description and not of MCVETO's DPG algorithms. If the TSL description of the x86 instruction set were augmented to incorporate the Interrupt Descriptor Table in the semantics, the YACC-like tool-generation capabilities would allow easy regeneration of augmented versions of the emulator and symbolic-analysis components used in MCVETO's DPG algorithm.

Moreover, the use of TSL aids the process of augmenting a system like MCVETO with non-standard semantic instrumentation that allows checking for policy violations. For instance, MCVETO currently uses a non-standard instrumented semantics in which the standard instruction-set semantics is augmented with an auxiliary stack [27, §3.5]. Initially, the auxiliary stack is empty; at each `call` instruction, a copy of the return address pushed on the processor stack is also pushed on the auxiliary stack; at each `ret` instruction, the auxiliary stack is checked to make sure that it is non-empty and that the address popped from the processor stack matches the address popped from the auxiliary stack.

6 Related Work

Machine-code analysis has been gaining increased attention, and by now there is a considerable literature on static, dynamic, and symbolic analysis of machine code. It includes such topics as platforms and infrastructure for performing analysis, improved methods to create CFGs, suitable abstract domains for dataflow

analysis of machine code, applications in software engineering and program understanding, verification of safety properties, testing (including discovery of security vulnerabilities), malware analysis, type inference, analysis of cache behavior, proof-carrying code, relating source code to the resulting compiled code, and low-level models of the semantics of high-level code. Space limitations preclude a detailed discussion of related work in this paper. An in-depth discussion of work related to CodeSurfer/x86 can be found in [3].

Acknowledgments

We are grateful to our collaborators at Wisconsin—A. Burton, E. Driscoll, M. Elder, and T. Andersen—and at GrammaTech, Inc.—T. Teitelbaum, D. Melski, S. Yong, T. Johnson, D. Gopan, and A. Loginov —for their many contributions to our work on machine-code analysis and verification.

References

1. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56 (2009)
2. Balakrishnan, G.: WYSINWYX: What You See Is Not What You eXecute. PhD thesis, C.S. Dept., Univ. of Wisconsin, Madison, WI, Tech. Rep. 1603 (August 2007)
3. Balakrishnan, G., Reps, T.: WYSINWYX: What You See Is Not What You eXecute. Trans. on Prog. Lang. and Syst. (to appear)
4. Balakrishnan, G., Reps, T.: Analyzing memory accesses in x86 executables. In: Comp. Construct., pp. 5–23 (2004)
5. Balakrishnan, G., Reps, T.: Recency-abstraction for heap-allocated storage. In: Static. Analysis Symp. (2006)
6. Balakrishnan, G., Reps, T.: DIVINE: DIscovering Variables IN Executables. In: Verif., Model Checking, and Abs. Interp. (2007)
7. Balakrishnan, G., Reps, T.: Analyzing stripped device-driver executables. In: Tools and Algs. for the Construct. and Anal. of Syst. (2008)
8. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.-H., Teitelbaum, T.: Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 158–163. Springer, Heidelberg (2005)
9. Balakrishnan, G., Reps, T., Melski, D., Teitelbaum, T.: WYSINWYX: What You See Is Not What You eXecute. In: Meyer, B., Woodcock, J. (eds.) VSTTE 2005. LNCS, vol. 4171, pp. 202–213. Springer, Heidelberg (2008)
10. Beckman, N., Nori, A., Rajamani, S., Simmons, R.: Proofs from tests. In: Int. Symp. on Softw. Testing and Analysis (2008)
11. Beyer, D., Henzinger, T., Majumdar, R., Rybalchenko, A.: Path invariants. In: Prog. Lang. Design and Impl. (2007)
12. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In: POPL (1977)
13. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. SCP 69(1-3) (2007)

14. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: Prog. Lang. Design and Impl. (2005)
15. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL (2010)
16. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: A new algorithm for property checking. In: Found. of Softw. Eng. (2006)
17. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
18. Johnson, S.: YACC: Yet another compiler-compiler. Technical Report Comp. Sci. Tech. Rep. 32, Bell Laboratories (1975)
19. Lal, A., Reps, T., Balakrishnan, G.: Extended weighted pushdown systems. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 434–448. Springer, Heidelberg (2005)
20. Lim, J., Lal, A., Reps, T.: Symbolic analysis via semantic reinterpretation. In: Păsăreanu, C.S. (ed.) SPIN Workshop. LNCS, vol. 5578, pp. 148–168. Springer, Heidelberg (2009)
21. Lim, J., Reps, T.: A system for generating static analyzers for machine instructions. In: Comp. Construct. (2008)
22. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS (2003)
23. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: European Symp. on Programming (2005)
24. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: POPL (1999)
25. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: Part. Eval. and Semantics-Based Prog. Manip. (2006)
26. Reps, T., Balakrishnan, G., Lim, J., Teitelbaum, T.: A next-generation platform for analyzing executables. In: Asian Symp. on Prog. Lang. and Systems (2005)
27. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 288–305. Springer, Heidelberg (2010)

Constraint Solving for Program Verification: Theory and Practice by Example

Andrey Rybalchenko

Technische Universität München

Abstract. Program verification relies on the construction of auxiliary assertions describing various aspects of program behaviour, e.g., inductive invariants, resource bounds, and interpolants for characterizing reachable program states, ranking functions for approximating number of execution steps until program termination, or recurrence sets for demonstrating non-termination. Recent advances in the development of constraint solving tools offer an unprecedented opportunity for the efficient automation of this task. This paper presents a series of examples illustrating algorithms for the automatic construction of such auxiliary assertions by utilizing constraint solvers as the basic computing machinery.

1 Introduction

Program verification has a long history of using constraint-based algorithms as main building blocks. In principle, constraint-based algorithms follow two major steps. First, during the constraint generation step a program property of interest is formulated as a set of constraints. Any solution to these constraints determines the property. During the second step, the constraints are solved. Usually, this step is executed using a separate constraint solving procedure. Such separation of concerns, i.e., constraint generation vs. solving, can liberate the designer of the verification tool from the tedious task of creating a dedicated algorithm. Instead, an existing off-the-shelf constraint solver can be put to work.

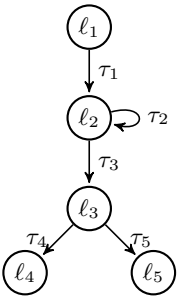
In this paper, we show how constraints can be used to prove program (non-)termination and safety by generating ranking functions, interpolants, invariants, resource bounds, and recurrence sets. First, we focus on assertions expressed in linear arithmetic, which form a practically important class, and then show extensions with uninterpreted function symbols. Our presentation uses a collection on examples to illustrate the algorithms.

The rest of the paper is organized as follows. Section 2 illustrates the generation of linear ranking functions. In Section 3, we show how linear interpolants can be computed. Section 4 presents linear invariant generation and an optimization technique that exploits program test cases. It also shows how invariant generation can be adapted to compute bounds on resource consumption. We use an additional, possibly non-terminating program in Section 5 to illustrate the construction of recurrence sets for proving non-termination. Section 6 shows how

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 57–71, 2010.

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x++;
  }
  assert(x >= z);
}
```

(a)



(b)

$$\rho_1 = (y \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_2 = (x + 1 \leq y \wedge x' = x + 1 \wedge y' = y \wedge z' = z)$$
$$\rho_3 = (x \geq y \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_4 = (x \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_5 = (x + 1 \leq z \wedge x' = x \wedge y' = y \wedge z' = z)$$

(c)

Fig. 1. An example program (a), its control-flow graph (b), and the corresponding transition relations (c)

constraint-based algorithms for the synthesis of linear assertions can be extended to deal with the combination of linear arithmetic and uninterpreted functions. Here, we use the interpolation algorithm as an example.

We use the program shown in Figure 1 as a source of termination, interpolation and safety proving obligations. When translating the program instructions into the corresponding transition relations we approximate integer program variables by rationals, in order to reduce the complexity the resulting constraint generation and solving tasks. Hence, the relation ρ_2 has a guard $x + 1 \leq y$. Furthermore, the failure of the assert statement is represented by reachability of the control location ℓ_5.

2 Linear Ranking Functions

Program termination is an important property that ensures its responsiveness. Proving program terminations requires construction of ranking functions that over-approximate the number of execution steps that the program can make from a given state until termination. Linear ranking functions express such approximations by linear assertions over the program variables.

Input. We illustrate the construction of ranking functions on the while loop from the program in Figure 1, as shown below. See [5] for its detailed description and pointers to the related work.

```
while (x < y) {
  x++;
}
```

We deliberately choose a loop that neither contains further nesting loops nor branching control flow inside the loop body in order to highlight the main ideas of the constraint-based ranking function generation.

Our algorithm will search for a linear expression over the program variables that proves termination. Such an expression is determined by the coefficients of the occurring variables. Let f_x and f_y be the coefficients for the variables x and y, respectively. Since the program variable z does not play a role in the loop, to simplify the presentation we do not take it into consideration.

A linear expression is a ranking function if its value is bounded from below for all states on which the loop can make a step, and is decreasing by some a priory fixed positive amount. Let δ_0 be the lower bound for the value of the ranking function, and δ by the lower bound on the amount of decrease. Then, we obtain the following defining constraint on the ranking function coefficients and the bound values.

$$\begin{aligned} &\exists f_x\ \exists f_y\ \exists \delta_0\ \exists \delta \\ &\quad \forall x\ \forall y\ \forall x'\ \forall y' : \\ &\qquad (\delta \geq 1 \wedge \\ &\qquad \rho_2 \rightarrow (f_x x + f_y y \geq \delta_0 \wedge \\ &\qquad\qquad f_x x' + f_y y' \leq f_x x + f_y y - \delta)) \end{aligned} \tag{1}$$

Any satisfying assignment to f_x, f_y, δ_0 and δ determines a linear ranking function for the loop.

The constraint (1) contains universal quantification over the program variables and their primed version, which makes it difficult to solve directly using existing constraint solvers. At the next step, we will address this obstacle by eliminating the universal quantification.

Constraints. First, we represent the transition relation of the loop in matrix form, which will help us during the constraint generation. After replacing equalities by conjunctions of corresponding inequalities, we obtain the matrix form below.

$$\begin{aligned} \rho_2 &= (x + 1 \leq y \wedge x' = x + 1 \wedge y' = y) \\ &= (x - y \leq -1 \wedge -x + x' \leq 1 \wedge x - x' \leq -1 \wedge -y + y' \leq 0 \wedge y - y' \leq 0) \\ &= \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \end{aligned}$$

The bound and decrease conditions from (1) produce the following matrix forms.

$$f_x x + f_y y \geq \delta_0 = \begin{pmatrix} -f_x & -f_y & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq -\delta_0$$

$$f_x x' + f_y y' \leq f_x x + f_y y - \delta = \begin{pmatrix} -f_x & -f_y & f_x & f_y \end{pmatrix} \begin{pmatrix} x \\ y \\ x' \\ y' \end{pmatrix} \leq -\delta$$

Now we are ready to eliminate the universal quantification. For this purpose we apply Farkas' lemma, which formally states

$$((\exists x : Ax \leq b) \wedge (\forall x : Ax \leq b \rightarrow cx \leq \gamma)) \leftrightarrow (\exists \lambda : \lambda \geq 0 \wedge \lambda A = c \wedge \lambda b \leq \gamma) \ .$$

This statement asserts that every linear consequence of a satisfiable set of linear inequalities can be obtained as a non-negative linear combination of these inequalities. As an immediate consequence we obtain that for a non-satisfiable set of linear inequalities we can derive an unsatisfiable inequality, i.e.,

$$(\forall x : \neg(Ax \leq b)) \leftrightarrow (\exists \lambda : \lambda \geq 0 \wedge \lambda A = 0 \wedge \lambda b \leq -1) \ .$$

By applying Farkas' lemma on (1) we obtain the following constraint.

$$\begin{array}{l} \exists f_x \ \exists f_y \ \exists \delta_0 \ \exists \delta \\ \quad \exists \lambda \ \exists \mu : \\ \qquad (\delta \geq 1 \wedge \\ \qquad \lambda \geq 0 \wedge \\ \qquad \mu \geq 0 \wedge \\ \qquad \lambda \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -f_x & -f_y & 0 & 0 \end{pmatrix} \wedge \lambda \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \leq -\delta_0 \wedge \\ \qquad \mu \begin{pmatrix} 1 & -1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} = \begin{pmatrix} -f_x & -f_y & f_x & f_y \end{pmatrix} \wedge \mu \begin{pmatrix} -1 \\ 1 \\ -1 \\ 0 \\ 0 \end{pmatrix} \leq -\delta \end{array} \tag{2}$$

This constraint contains only existentially quantified rational variables and consists of linear (in)equalities. Thus, it can be efficiently solved by the existing tools for Linear Programming over rationals.

Solution. We apply a linear constraint solver on (2) and obtain the following solution.

$$\begin{aligned}\lambda &= (1\,0\,0\,0\,0)\\ \mu &= (0\,0\,1\,1\,0)\\ f_x &= -1\\ f_y &= 1\\ \delta_0 &= 1\\ \delta &= 1\end{aligned}$$

This solution states that the expression $-x + y$ decreases during each iteration of the loop by at least 1, and is greater than 1 for all states that satisfy the loop guard.

3 Constraint Linear Interpolants

Interpolants are logical assertions over program states that can separate program states that satisfy a desired property from the ones that violate the property. Interpolants play an important role in automated abstraction of sets of program states and their automatic construction is a crucial building block for program verification tools. In this section we present an algorithm for the computation of linear interpolants. A unique feature of our algorithm is the possibility to bias the outcome using additional constraints.

In program verification, interpolants are computed for formulas that are extracted from program paths, i.e., sequences of program statements that follow the control flow graph of the program. We illustrate the interpolant computation algorithm using a program path from Figure 1, and refer to [7] for a detailed description of the algorithm and a discussion of the related work.

Input. We consider a path $\tau_1\tau_3\tau_5$, which corresponds to an execution of the program that does not enter the loop and fails the assert statement. This path does not modify the values of the program variables, but rather imposes a sequence of conditions $y \geq z \wedge x \geq y \wedge x + 1 \leq z$. Since this sequence is not satisfiable, a program verifier can issue an interpolation query that needs to compute a separation between the states that the program reaches after taking the transition τ_3 and the states that violate the assertion. Formally, we are interested in an inequality $i_x x + i_y y + i_z z \leq i_0$, called an interpolant, such that

$$\begin{aligned}&\exists i_x\ \exists i_y\ \exists i_z\ \exists i_0\\ &\quad \forall x\ \forall y\ \forall z\ :\\ &\qquad ((y \geq z \wedge x \geq y) \rightarrow i_x x + i_y y + i_z z \leq i_0) \wedge\\ &\qquad ((i_x x + i_y y + i_z z \leq i_0 \wedge x + 1 \leq z) \rightarrow 0 \leq -1)\end{aligned} \tag{3}$$

Furthermore, we require that $i_x x + i_y y + i_z z \leq i_0$ only refers to the variables that appear both in $y \geq z \wedge x \geq y$ and $x + 1 \leq z$, which are x and z. Hence, i_z needs to be equal to 0, which is ensured by the above constraint without any additional effort.

Constraints. First we represent the sequence of conditions in matrix form as follows.

$$
\begin{aligned}
&(y \geq z \wedge x \geq y \wedge x + 1 \leq z) = \\
&(-y + z \leq 0 \wedge -x + y \leq 0 \wedge x - z \leq -1) = \\
&\begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}
\end{aligned}
$$

Since (3) contains universal quantification, we apply Farkas' to enable applicability of Linear Programming tools and obtain the following constraint.

$$
\begin{aligned}
&\exists i_x\ \exists i_y\ \exists i_z\ \exists i_0 \\
&\quad \exists \lambda\ \exists \mu : \\
&\qquad \lambda \geq 0 \wedge \mu \geq 0 \wedge \\
&\qquad (\lambda\ \mu) \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \\ 1 & 0 & -1 \end{pmatrix} = 0 \wedge (\lambda\ \mu) \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix} \leq -1 \wedge \\
&\qquad (i_x\ i_y\ i_z) = \lambda \begin{pmatrix} 0 & -1 & 1 \\ -1 & 1 & 0 \end{pmatrix} \wedge i_0 = \lambda \begin{pmatrix} 0 \\ 0 \end{pmatrix}
\end{aligned}
\tag{4}
$$

This constraint uses two vectors λ and μ to represent the linear combination that derives the unsatisfiable inequality $0 \leq -1$. The vector λ tracks the first two inequalities, and μ tracks the third inequality.

Solution. By solving (4) we obtain

$$
\begin{aligned}
\lambda &= (1\ 1), \\
\mu &= 1\ , \\
i_x &= -1\ , \\
i_y &= 0\ , \\
i_z &= 1\ , \\
i_0 &= 0\ .
\end{aligned}
$$

The resulting interpolant is $-x + z \leq 0$.

The constraint-based approach to interpolant computation offers a unique opportunity to bias the resulting interpolant using additional constraints. That is, (4) can be extended with an additional constraint $C\binom{i}{i_0} \leq c$ that encode the bias condition.

4 Linear Invariants

Invariants are assertions over program variables whose value does not change during program execution. In program verification invariants are used to describe sets of reachable program states, and are an indispensable tool for reasoning about program correctness. In this section, we show how invariants proving the non-reachability of the error location in the program can be computed by using constraint-based techniques, and present a testing-based approach for simplifying the resulting constraint generation task. Furthermore, we briefly present

a close connection between invariant and bound generation. See [4,2] for the corresponding algorithms and further details.

We illustrate the invariant generation on the program shown in Figure 1 and construct an invariant that proves the non-reachability of the location ℓ_5, which serves as the error location.

Input. Our goal is to compute two linear inequalities over program variables $p_x x + p_y y + p_z z \leq p_0$ and $q_x x + q_y y + q_z z \leq q_0$ for the locations ℓ_2 and ℓ_3, respectively, such that these inequalities (1) represent all program states that are reachable at the respective locations, (2) serve as an induction hypothesis for proving (1) by induction over the number of program steps required to reach a program state, and (3) imply that no program execution can reach the error location ℓ_5. We encode the conditions (1–3) on the unknown invariant coefficients as the following constraint.

$$\begin{aligned}
&\exists p_x\ \exists p_y\ \exists p_z\ \exists p_0\ \exists q_x\ \exists q_y\ \exists q_z\ \exists q_0 \\
&\quad \forall x\ \forall y\ \forall z\ \forall x'\ \forall y'\ \forall z' : \\
&\qquad (\rho_1 \rightarrow p_x x' + p_y y' + p_z z' \leq p_0) \wedge \\
&\qquad ((p_x x + p_y y + p_z z \leq p_0 \wedge \rho_2) \rightarrow p_x x' + p_y y' + p_z z' \leq p_0) \wedge \\
&\qquad ((p_x x + p_y y + p_z z \leq p_0 \wedge \rho_3) \rightarrow q_x x' + q_y y' + q_z z' \leq q_0) \wedge \\
&\qquad ((q_x x + q_y y + q_z z \leq p_0 \wedge \rho_4) \rightarrow 0 \leq 0) \wedge \\
&\qquad ((q_x x + q_y y + q_z z \leq p_0 \wedge \rho_5) \rightarrow 0 \leq -1)
\end{aligned} \tag{5}$$

For each program transition this constraint contains a corresponding conjunct. The conjunct ensures that given a set of states at the start location of the transition all states reachable by applying the transition are represented by the assertion associated with the destination location. For example, the first conjunct asserts that applying τ_1 on any state leads to a state represented by $p_x x + p_y y + p_z z \leq p_0$.

Constraints. Since (5) contains universal quantification, we resort to the Farkas' lemma-based elimination, which yields the following constraint.

$$\begin{aligned}
&\exists p_x\ \exists p_y\ \exists p_z\ \exists p_0\ \exists q_x\ \exists q_y\ \exists q_z\ \exists q_0 \\
&\quad \exists \lambda_1\ \exists \lambda_2\ \exists \lambda_3\ \exists \lambda_4\ \exists \lambda_5 : \\
&\qquad \lambda_1 \geq 0 \wedge \cdots \wedge \lambda_5 \geq 0 \wedge \\
&\qquad \lambda_1 R_1 = (0\ p_x\ p_y\ p_z) \wedge \lambda_1 r_1 \leq p_0 \wedge \\
&\qquad \lambda_2 \begin{pmatrix} p_x\ p_y\ p_z\ 0 \\ R_2 \end{pmatrix} = (0\ p_x\ p_y\ p_z) \wedge \lambda_2 \begin{pmatrix} p_0 \\ r_2 \end{pmatrix} \leq p_0 \wedge \\
&\qquad \lambda_3 \begin{pmatrix} p_x\ p_y\ p_z\ 0 \\ R_3 \end{pmatrix} = (0\ q_x\ q_y\ q_z) \wedge \lambda_3 \begin{pmatrix} p_0 \\ r_3 \end{pmatrix} \leq q_0 \wedge \\
&\qquad \lambda_4 \begin{pmatrix} q_x\ q_y\ q_z\ 0 \\ R_4 \end{pmatrix} = 0 \wedge \lambda_4 \begin{pmatrix} q_0 \\ r_4 \end{pmatrix} \leq 0 \wedge \\
&\qquad \lambda_5 \begin{pmatrix} q_x\ q_y\ q_z\ 0 \\ R_5 \end{pmatrix} = 0 \wedge \lambda_5 \begin{pmatrix} q_0 \\ r_5 \end{pmatrix} \leq -1
\end{aligned} \tag{6}$$

Unfortunately, this constraint is non-linear since it contains multiplication between unknown components of $\lambda_1, \ldots, \lambda_5$ and the unknown coefficients $p_x, p_y, p_z, p_0, q_x, q_y, q_z, q_0$.

Solution. In contrast to interpolation or ranking function generation, we cannot directly apply Linear Programming tools to solve (6) and need to introduce additional solving steps, as described in Section 4.1 and [4]. These steps lead to the significant reduction of the number of non-linear terms, and make the constraints amenable to solving using case analysis on the remaining unknown coefficients for derivations.

For our program we obtain the following solution.

$$\begin{array}{llll} & \lambda_1 = (1\ 1\ 1\ 1) & & \\ & \lambda_2 = (1\ 0\ 1\ 1\ 1) & & \\ & \lambda_3 = (1\ 1\ 1\ 1\ 1) & & \\ & \lambda_4 = (0\ 0\ 0\ 0\ 0) & & \\ & \lambda_5 = (1\ 1\ 0\ 0\ 0) & & \\ p_x = 0 & p_y = -1 & p_z = 1 & p_0 = 0 \\ q_x = -1 & q_y = 0 & q_z = 1 & q_0 = 0 \end{array}$$

This solution defines an invariant $-y + x \leq 0$ at the location ℓ_2 and $-x + z \leq 0$ at the location ℓ_3.

4.1 Static and Dynamic Constraint Simplification

Now we show how program test cases can be used to obtain additional constraint simplification when computing invariants.

We use the program in Figure 1 and consider a set of program states below, which can be recorded during a test run of the program.

$$\begin{array}{l} s_1 = (\ell_1, x = 1, y = 0, z = 2) \\ s_2 = (\ell_2, x = 2, y = 0, z = 2) \\ s_3 = (\ell_2, x = 2, y = 1, z = 2) \\ s_4 = (\ell_2, x = 2, y = 1, z = 2) \\ s_5 = (\ell_2, x = 2, y = 2, z = 2) \\ s_6 = (\ell_3, x = 3, y = 2, z = 2) \end{array}$$

These states are reachable, hence any program invariant holds for these states. Hence, we perform a partial evaluation of the unknown invariant templates at locations ℓ_2 and ℓ_3 on states $s_2, \ldots, s_5$ and s_6, respectively:

$$\begin{array}{l} \varphi_1 = (p_x 1 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_2 = (p_x 1 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_3 = (p_x 2 + p_y 2 + p_z 1 \leq p_0) \\ \varphi_4 = (q_x 2 + q_y 2 + q_z 1 \leq q_0) \end{array}$$

The obtained constraints are linear and they must hold for any template instantiation. Hence the constraint

$$p_x + 2p_y + p_z \leq p_0 \wedge p_x + 2p_y + p_z \leq p_0 \wedge 2p_x + 2p_y + p_z \leq p_0 \wedge 2q_x + 2q_y + q_z \leq q_0$$

can be added to (6) as an additional strengthening without changing the set of solutions. Practically however this strengthening results in a series of simplifications of the non-linear parts of (6), which can dramatically increase the constraint solving efficiency.

4.2 Bound Generation as Unknown Assertion

Program execution can consume various resources, e.g., memory or time. For our example program in Figure 1, the number of loop iterations might be such a resource since it correlates with the program execution time. Resource bounds are logical assertions that provide an estimate on the resource consumption, and their automatic generation is an important task, esp. for program execution environments with limited resource availability.

There is a close connection between expressions describing resource bounds and program assertions specifying conditions on reachable program states. We can encode the check if a given bound holds for all program execution as a program assertion over auxiliary program variables that keep track of the resource consumption. In our example the assertion statement ensures that the consumption of time, as tracked by the variable x, is bounded from above by the value of the variable z.

Unknown resource bounds can be synthesized using our constraint-based invariant generation algorithm described above after a minor modification of the employed constraint encoding. Next we show how to modify our constraints (5) and (6) to identify a bound on the number of loop iterations, under the assumption that the assertion statement is not present in the program.

First, we assume that the unknown bound assertion is represented by an inequality

$$x \leq b_y y + b_z z + b_0 \ .$$

Now, our goal is to identify the values of the coefficients b_y, b_z, and b_0 together with an invariant that proves the validity of the bound.

We encode our goal as a constraint by replacing the last conjunct in (5), which was present due to the assertion statement, with the following implication.

$$q_x x + q_y y + q_z z \leq q_0 \rightarrow x \leq b_y \leq b_y y + b_z z + b_0$$

This implication requires that the program invariant at the location after the loop exit implies the bound validity.

After eliminating universal quantification from the modified constraint and a subsequent solving attempt we realize that no bound on x can be found. If we consider a modified program that includes an assume statement

```
assume(z>=x);
```

as its first instruction and reflect the modification in the constraints, then we will be able to compute the following bound.

$$x \leq y$$

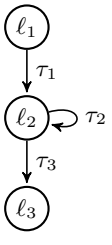
5 Recurrence Sets

Inherent limitations of the existing tools for proving program termination can lead to cases when non-conclusive results are reported. Since a failure to find a termination argument does not directly imply that the program does not terminate on certain inputs, we need dedicated methods that can prove non-termination of programs. In this section we present such a method. It is based on the notion of recurrence set that serves as a proof for the existence of a non-terminating program execution.

Input. We show how non-termination can be proved by constructing recurrence sets using the example in Figure 2. The complete version of the corresponding algorithm is presented in [3].

```
main(int x, int y, int z) {
  assume(y >= z);
  while (x < y) {
    x=x+1+z;
  }
}
```

(a)



(b)

$$\rho_1 = (y \geq z \wedge x' = x \wedge y' = y \wedge z' = z)$$
$$\rho_2 = (x + 1 \leq y \wedge x' = x + 1 + z \wedge y' = y \wedge z' = z)$$
$$\rho_3 = (x \geq y \wedge x' = x \wedge y' = y \wedge z' = z)$$

(c)

Fig. 2. A non-terminating example program (a), its control-flow graph (b), and the corresponding transition relations (c)

To prove non-termination we will compute a recurrence set consisting of program states that can be reached at the loop entry and lead to an additional loop iteration. We assume that a desired recurrence set can be expressed by a conjunction of two inequalities $pv \leq p_0 \wedge qv \leq q_0$ over the vector of program variables v consisting of x, y, and z, while p, p_0, q, and q_0 are unknown coefficients. To simplify notation, we write $Sv \leq s$ for the conjunction of $pv \leq p_0$ and $qv \leq q_0$. Then, the following constraint encodes the recurrence set condition.

$$\begin{aligned}
&\exists S\ \exists s : \\
&\quad (\exists v : Sv \leq s) \wedge \\
&\quad (\exists v\ \exists v' : \rho_1(v, v') \wedge Sv' \leq s) \wedge \\
&\quad (\forall v\ \exists v' : Sv \leq s \rightarrow (\rho_2(v, v') \wedge Sv' \leq s))
\end{aligned} \tag{7}$$

The first conjunct guarantees that the recurrence set is not empty. The second conjunct requires that the recurrence set contains at least one state that is reachable by following the transition τ_1, i.e., by when the loop is reached for the first time. The last conjunct guarantees that every state in the recurrence set can follow the loop transition τ_2 and get back to the recurrence set. Together, these properties guarantee that there exists an infinite program execution that can be constructed from the elements of the recurrence set.

Constraints. The constraint (7) contains universal quantification and quantifier alternation, which makes it difficult to solve using the existing quantifier elimination tools. As the first step, we simplify (7) by exploiting the structure of transition relations ρ_1 and ρ_2. Our simplification relies on the fact that the values of primed variables are defined by update expressions, and substitutes the primed variables by the corresponding update expressions. We obtain the following simplified equivalent of (7).

$$\begin{aligned}
&\exists S\ \exists s : \\
&\quad (\exists x\ \exists y\ \exists z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists x\ \exists y\ \exists z : y \geq z \wedge S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\forall x\ \forall y\ \forall z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s \rightarrow (x + 1 \leq y \wedge S \begin{pmatrix} x+1+z \\ y \\ z \end{pmatrix} \leq s))
\end{aligned}$$

Furthermore, we will omit the first conjunct since it is subsumed by the second conjunct.

Next, we eliminate universal quantification by applying Farkas' lemma. The application yields the following constraint. It only contains existential quantification and uses S_x, S_y, and S_z to refer to the first, second, and the third column of S, respectively.

$$\begin{aligned}
&\exists S\ \exists s : \\
&\quad (\exists x\ \exists y\ \exists z : S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists x\ \exists y\ \exists z : y \geq z \wedge S \begin{pmatrix} x \\ y \\ z \end{pmatrix} \leq s) \wedge \\
&\quad (\exists \lambda : \lambda \geq 0 \wedge \lambda S = \begin{pmatrix} 1 & -1 & 0 \end{pmatrix} \wedge \lambda s \leq -1) \wedge \\
&\quad (\exists \Lambda : \Lambda \geq 0 \wedge \Lambda S = \begin{pmatrix} S_x & S_y & S_z + S_x \end{pmatrix} \wedge \Lambda s \leq (s - S_x))
\end{aligned} \tag{8}$$

Solution. We apply solving techniques that we used for dealing with non-linear constraints during invariant generation, see Section 4, and obtain the following solution.

$$
\begin{aligned}
x &= -2 \\
y &= -1 \\
z &= -1 \\
\lambda &= (1\ 0) \\
\Lambda &= \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \\
p &= (1\ -1\ 0) \\
p_0 &= -1 \\
q &= (0\ 0\ 1) \\
q_0 &= -1
\end{aligned}
$$

This solution defines the recurrence set

$$x - y \leq -1 \wedge z \leq -1 \ ,$$

and states that the program does not terminate if executed from an initial state that assigns $x = -2$, $y = -1$, and $z = -1$.

6 Combination with Uninterpreted Functions

In the previous sections we showed how auxiliary assertions represented by linear inequalities can be generated using constraint-based techniques. In this section we show that these techniques can be directly extended to deal with assertions represented by linear arithmetic combined with uninterpreted functions. This combined theory plays in important role in program verification, where uninterpreted functions are used to abstract functions that too complex to be modeled precisely. The basis of the extension is the hierarchical approach to the combination of logical theories [6]. We refer to [7,1] for constraint-based interpolation and invariant generation algorithms for the combination of linear arithmetic and uninterpreted functions. Next, we will illustrate the interpolation algorithm for linear arithmetic and function symbols using a small example.

Input. The interpolation algorithm takes as input a pair of mutually unsatisfiable assertions φ and ψ shown below.

$$
\begin{aligned}
\varphi &= (x \leq a \wedge a \leq y \wedge f(a) \leq 0) \\
\psi &= (y \leq b \wedge b \leq x \wedge 1 \leq f(b))
\end{aligned}
$$

The proof of unsatisfiability requires reasoning about linear arithmetic and uninterpreted function, which we represent by the logical consequence relation $\models_{\text{LI+UIF}}$.

$$\varphi \wedge \psi \models_{\text{LI+UIF}} \bot$$

The goal of the interpolation algorithm is to construct an assertion χ such that

$$\begin{aligned} &\varphi \models_{\text{LI+UIF}} \chi \ , \\ &\chi \wedge \psi \models_{\text{LI+UIF}} \bot \ , \\ &\chi \text{ is expressed over common symbols of } \varphi \text{ and } \psi \ . \end{aligned} \tag{9}$$

Constraints and solution. As common in reasoning about combined theories, we first apply a purification step that separates arithmetic constraints from the function applications as follows.

$$\begin{aligned} \varphi_{\text{LI}} = \ & (x \leq a \wedge a \leq y \wedge c \leq 0) \\ \psi_{\text{LI}} = \ & (y \leq b \wedge b \leq x \wedge 1 \leq d) \\ D = \ & \{c \mapsto f(a), d \mapsto f(b)\} \\ X = \ & \{a = b \rightarrow c = d\} \end{aligned}$$

The sets of inequalities φ_{LI} and ψ_{LI} do not have any function symbols, which were replaced by fresh variables. The mapping between these fresh variables and the corresponding function applications is give by the set D. The set X contains functionality axiom instances that we create for all pairs of occurrences of function applications. These instances are expressed in linear arithmetic. For our example there is only one such instance.

The hierarchical reasoning approach guarantees that instances collected in X are sufficient for proving the mutual unsatisfiability of the pure assertions φ_{LI} and ψ_{LI}, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \wedge \bigwedge X \models_{\text{LI}} \bot$$

Unfortunately we cannot apply an algorithm for interpolation in linear arithmetic on the unsatisfiable conjunction presented above since the axiom instance in X contains variables that appear both in φ_{LI} and ψ_{LI}, which will lead to an interpolation result that violates the third condition in (9).

Instead, we resort to a case-based reasoning as follows. First, we attempt to compute an interpolant by considering the pure assertions, but do not succeed since they are mutually satisfiable, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \not\models_{\text{LI}} \bot$$

Nevertheless, the conjunction of pure assertions implies the precondition for applying the functionality axiom instance from X, i.e.,

$$\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a = b$$

From this implication follows that we can compute intermediate terms that are represented over variables that are common to φ_{LI} and ψ_{LI}. Formally, we have

$$\begin{aligned} &\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a \leq y \wedge y \leq b \ , \\ &\varphi_{\text{LI}} \wedge \psi_{\text{LI}} \models_{\text{LI}} a \geq x \wedge x \geq b \ . \end{aligned}$$

We rearrange these implications and obtain the following implications.

$$\varphi_{\mathrm{LI}} \models_{\mathrm{LI}} x \leq a \wedge a \leq y$$
$$\psi_{\mathrm{LI}} \models_{\mathrm{LI}} y \leq b \wedge b \leq x$$

These implications are used by our interpolation algorithm to derive appropriate case reasoning, which will be presented later on. Furthermore, our algorithm creates an additional function application $f(y)$ together with a corresponding fresh variable e, which is used for the purification and is recorded in the set D.

$$D \;=\; \{c \mapsto f(a), d \mapsto f(b), e \mapsto f(y)\}$$

The first step of the case reasoning requires computing an interpolant for the following unsatisfiable conjunction.

$$(\varphi_{\mathrm{LI}} \wedge a = e) \wedge (\psi_{\mathrm{LI}} \wedge e = b) \models_{\mathrm{LI}} \bot$$

By applying the algorithm presented in Section 3 we obtain a partial interpolant $e \leq 0$ such that

$$\varphi_{\mathrm{LI}} \wedge a = e \models_{\mathrm{LI}} e \leq 0 \;,$$
$$e \leq 0 \wedge \psi_{\mathrm{LI}} \wedge e = b \models_{\mathrm{LI}} \bot \;.$$

The partial interpolant is completed using the case reasoning information as follows.

$$\chi_{\mathrm{LI}} \;=\; (x \neq y \vee (x = y \wedge e \leq 0))$$

After replacing the fresh variables by the corresponding function applications we obtain the following interpolant χ for the original input φ and ψ.

$$\begin{aligned}\chi = \;& (x \neq y \vee (x = y \wedge e \leq 0))[f(q)/e] \\ =\; & x \neq y \vee (x = y \wedge f(q) \leq 0)\end{aligned}$$

7 Conclusion

We presented a collection of examples demonstrating that several kinds of auxiliary assertions that play a crucial role in program verification can be effectively synthesized using constraint-based techniques.

Acknowledgment. I thank Byron Cook, Fritz Eisenbrand, Ashutosh Gupta, Tom Henzinger, Rupak Majumdar, Andreas Podelski, and Viorica Sofronie-Stokkermans for unconstrained satisfactory discussions.

References

1. Beyer, D., Henzinger, T.A., Majumdar, R., Rybalchenko, A.: Invariant synthesis for combined theories. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 378–394. Springer, Heidelberg (2007)
2. Cook, B., Gupta, A., Magill, S., Rybalchenko, A., Simsa, J., Singh, S., Vafeiadis, V.: Finding heap-bounds for hardware synthesis. In: FMCAD (2009)

3. Gupta, A., Henzinger, T.A., Majumdar, R., Rybalchenko, A., Xu, R.-G.: Proving non-termination. In: POPL (2008)
4. Gupta, A., Majumdar, R., Rybalchenko, A.: From tests to proofs. In: TACAS (2009)
5. 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)
6. 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)
7. Sofronie-Stokkermans, V., Rybalchenko, A.: Constraint solving for interpolation. J. of Symbolic Computation (to appear, 2010)

Invariant Synthesis for Programs Manipulating Lists with Unbounded Data*

Ahmed Bouajjani[1], Cezara Drăgoi[1], Constantin Enea[1],
Ahmed Rezine[2], and Mihaela Sighireanu[1]

[1] LIAFA, University of Paris Diderot and CNRS, 75205 Paris 13, France
{abou,cezarad,cenea,sighirea}@liafa.jussieu.fr
[2] Uppsala University, Sweden
rahmed@it.uu.se

Abstract. We address the issue of automatic invariant synthesis for sequential programs manipulating singly-linked lists carrying data over infinite data domains. We define for that a framework based on abstract interpretation which combines a specific finite-range abstraction on the shape of the heap with an abstract domain on sequences of data, considered as a parameter of the approach. We instantiate our framework by introducing different abstractions on data sequences allowing to reason about various aspects such as their sizes, the sums or the multisets of their elements, or relations on their data at different (linearly ordered or successive) positions. To express the latter relations we define a new domain whose elements correspond to an expressive class of first order universally quantified formulas. We have implemented our techniques in an efficient prototype tool and we have shown that our approach is powerful enough to generate non-trivial invariants for a significant class of programs.

1 Introduction

Invariant synthesis is an essential ingredient in various program verification and analysis methodologies. In this paper, we address this issue for sequential programs manipulating singly-linked lists carrying data over infinite data domains such as integers or reals. Specifications of such programs typically involve constraints on various aspects such as the sizes of the lists, the multisets of their elements, as well as relations between data at their different positions, e.g., ordering constraints or even more complex arithmetical constraints on consecutive elements, or combining relations between the sizes, the sum of all elements, etc., of different lists.

Consider for instance the procedure `Dispatch3` given in Figure 1(b). It puts all the cells of the input list which have data larger than 3 to the list `grt`, and it puts all the other ones to the list `less`. Naturally, the specification of this procedure (at line 10) includes (1) the property expressed by the universally quantified first-order formula

$$\forall y.\ \mathtt{grt} \xrightarrow{*} y \Rightarrow \mathtt{data}(y) \geq 3 \quad \wedge \quad \forall y.\ \mathtt{less} \xrightarrow{*} y \Rightarrow \mathtt{data}(y) < 3 \qquad \text{(A)}$$

* This work was partly supported by the French National Research Agency (ANR) projects Averiss (ANR-06-SETIN-001) and Veridyc (ANR-09-SEGI-016).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 72–88, 2010.

which says that every cell y reachable from grt (resp. less) have data greater (resp. smaller) than 3, and (2) the preservation property saying that the multiset of the input list is equal to the union of the multisets of the two output lists. This property is expressed by

$$\texttt{ms\_init} = \texttt{ms}(\texttt{grt}) \cup \texttt{ms}(\texttt{less}) \tag{B}$$

where the variable ms_init represents the multiset of the elements of the input list, and ms(grt) (resp. ms(less)) denotes the multiset of the elements of grt (resp. less). A weaker property is length preservation, expressed by:

$$\texttt{len\_init} = \texttt{len}(\texttt{grt} \xrightarrow{+} \texttt{null}) + \texttt{len}(\texttt{less} \xrightarrow{+} \texttt{null}), \tag{C}$$

where len_init is the length of the input list.

```
   procedure Fibonacci(list* head)
1: { list *x=head;
2:   int m1=1;
3:   int m2=0;
4:   while (x != null)
5:   { x->data=m1+m2;
6:     m1=m2;
7:     m2=x->data;
8:     x=x->next;
9:   }
10:}
```

(a)

```
   procedure Dispatch3(list* head)
1: { list *tmp=null, grt=null, less=null;
2:   while (head != null)
3:   { tmp=head->next;
4:     if (head->data >= 3)
5:     { head->next=grt; grt=head; }
6:     else
7:     { head->next=less; less=head; }
8:    head=tmp;
9:   }
10:}
```

(b)

Fig. 1. Procedures Fibonacci and Dispatch3

The specification of sorting algorithms is similar since it includes an ordering constraint on the output list that is easily expressible using a universally quantified first-order formula, and a preservation constraint saying that the input and output lists have the same elements that is expressible using multiset constraints.

Moreover, an interesting property of the procedure Dispatch3 above is that the sum of all the elements in the list grt is larger than 3 times the size of that list, i.e.

$$\sum_{\texttt{grt} \xrightarrow{*} y} \texttt{data}(y) - 3 \times \texttt{len}(\texttt{grt} \xrightarrow{+} \texttt{null}) \geq 0 \tag{D}$$

Consider now the procedure Fibonacci given in Figure 1(a). It takes a list as an input and initializes its elements following the Fibonacci sequence. The natural specification for the procedure (at line 10) is expressed by the universally-quantified formula

$$\forall y_1, y_2, y_3.\ \texttt{head} \xrightarrow{*} y_1 \rightarrow y_2 \rightarrow y_3 \Rightarrow \texttt{data}(y_3) = \texttt{data}(y_2) + \texttt{data}(y_1) \tag{E}$$

which corresponds precisely to the definition of the Fibonacci sequence. Moreover, an interesting property of the Fibonacci sequence $\{f_i\}_{i \geq 1}$ is that $\sum_{i=1}^{i=n} f_i = 2f_n + f_{n-1} - 1$. This can be expressed (again at line 10) by the following constraint

$$\sum_{\texttt{head} \xrightarrow{*} y} \texttt{data}(y) = 2 \times \texttt{m2} + \texttt{m1} - 1 \tag{F}$$

The automatic synthesis of invariants like those shown above is a challenging problem since it requires combining in a nontrivial way different analysis techniques. This paper introduces a uniform framework based on abstract interpretation for tackling this problem. We define a generic abstract domain $\mathcal{A}_{\mathbb{HS}}$ for reasoning about dynamic lists with unbounded data which includes an abstraction on the shape of the heap and which is parametrized by some abstract domain on finite sequences of data (a data words abstract domain, $\mathbb{DW}$-domain for short). The latter is intended to abstract the sequences of data in the lists by capturing relevant aspects such as their sizes, the sums or the multisets of their elements, or some class of constraints on their data at different (linearly ordered or successive) positions.

We instantiate our framework by defining new $\mathbb{DW}$-domains corresponding to the aspects mentioned above. The most complex $\mathbb{DW}$-domain is composed of first-order formulas such that their (quantified) universal part is of the form $\forall \mathbf{y}.\ (P \Rightarrow U)$, where $\mathbf{y}$ is a vector of variables interpreted to positions in the words, P is a constraint on the positions (seen as integers) associated with the $\mathbf{y}$'s, and U is a constraint on the data values at these positions, and possibly also on the positions when data are of numerical type. Then, we assume that our $\mathbb{DW}$-domain on first-order properties is parametrized by some abstract data domain, and we consider that U is defined as an object in that abstract domain. For the sake of simplicity of the presentation, we consider in the rest of the paper that the data are always of type integer (and therefore it is possible to take as abstract data domains the standard octagons or polyhedra abstract domains for instance). Our approach can in fact be applied to any other data domain. As for the syntax of the constraint P, we assume that we are given a finite set of fixed patterns (or templates) such as, for instance, order constraints or difference constraints.

Then, an object in the domain $\mathcal{A}_{\mathbb{HS}}$ is a finite collection of pairs $(\widetilde{G}, \widetilde{W})$ such that (1) $\widetilde{G}$ is a graph (where each node has an out-degree of at most 1) representing the set of all the garbage-free heap graphs that can be obtained by inserting sequences of non-shared nodes (nodes with in-degree 1) between any pair of nodes in $\widetilde{G}$ (thus edges in $\widetilde{G}$ represents list segments without sharing), and (2) $\widetilde{W}$ is an abstract object in the considered $\mathbb{DW}$-domain constraining the sequences of data attached to each edge in $\widetilde{G}$. So, all the shared nodes in the concrete heaps are present in $\widetilde{G}$, but $\widetilde{G}$ may have nodes which are not shared. Non-shared nodes which are not pointed by program variables are called simple nodes. We assume that objects in our abstract domain have graphs with k simple nodes, for some given bound k that is also a parameter of the domain. This assumption implies that the number of such graphs is finite (since for a given program with lists it is well known that the number of shared nodes is bounded).

We define sound abstract transformers for the statements in the class of programs we consider. Due to the bound on the number of simple nodes, and since heap transformations may add simple nodes, we use a normalization operation that shrinks paths of simple nodes into a single edge. This operation is accompanied with an operation that generalizes the known relations on the data attached to the eliminated simple nodes in order to produce a constraint (in the $\mathbb{DW}$-domain) on the data word associated with the edge resulting from the normalization. This step is actually quite delicate and special care has to be taken in order to keep preciseness. In particular, this is the crucial step that allows to generate universally quantified properties from a number of relations

between a finite (bounded) number of nodes. We have defined sufficient conditions on the sets of allowed patterns under which we obtain best abstract transformers.

We have implemented (in C) a prototype tool CINV based on our approach, and we have carried out several experiments (more than 30 examples) on list manipulating programs (including for instance sorting algorithms such as insertion sort, and the two examples in Figure 1).

2 Modeling and Reasoning about Programs with Singly-Linked Lists

We consider a class of strongly typed imperative programs manipulating dynamic singly linked lists. We suppose that all manipulated lists have the same type, i.e., reference to a record called `list` including one reference field `next` and one data field `data` of integer type. While the generalization to records with several data fields is straightforward, the presence of a single reference field is important for this work. The programs we consider do not contain procedure calls or concurrency constructs.

Program syntax: Programs are defined on a set of data variables *DVar* of type $\mathbb{Z}$ and a set of pointer variables *PVar* of type `list` (which includes the constant `null`). Data variables can be used in *data terms* built using operations over $\mathbb{Z}$ and in boolean conditions on data built using predicates over $\mathbb{Z}$. Pointers can be used in data terms (`p->data` and in assignments corresponding to heap manipulation like memory allocation/deallocation (`new`/`free`), selector field updates (`p->next=...`, `p->data=...`), and pointer assignments (`p=...`). Boolean conditions on pointers are built using predicates (`p==q` and `p==null`) testing for equality and definedness of pointer variables. No arithmetics is allowed on pointers. We allow sequential composition (`;`), conditionals (`if-then-else`), and iterations (`while`). The full syntax is given in [2].

Program semantics: A program configuration is given by a configuration for the heap and a valuation of data variables. Heaps can be represented naturally by a directed graph. Each object of type `list` is represented by a node. The constant `null` is represented by a distinguished node $\sharp$. The pointer field `next` is represented by the edges of the graph. The nodes are labeled by the values of the data field `data` and by the program pointer variables which are pointing to the corresponding objects. Every node has exactly one successor, except for $\sharp$, the node representing `null`. For example, the graph in Figure 4(a) represents a heap containing two lists $[4,0,5,2,3]$ and $[1,4,3,6,2,3]$ which share their two last cells. Two of the nodes are labeled by the pointer variables x and y.

Definition 1. *A* heap over *PVar* and *DVar is a tuple* $H = (N,S,V,L,D)$ *where:*
– N *is a finite set of nodes which contains a distinguished node* $\sharp$,
– $S : N \rightharpoonup N$ *is a successor partial function s.t. only* $S(\sharp)$ *is undefined,*
– $V : PVar \to N$ *is a function associating nodes to pointer variables s.t.* $V(\texttt{null}) = \sharp$,
– $L : N \rightharpoonup \mathbb{Z}$ *is a partial function associating nodes to integers s.t. only* $L(\sharp)$ *is undefined,*
– $D : DVar \to \mathbb{Z}$ *is a valuation for the data variables.*

A node which is labeled by a pointer variable or which has at least two predecessors is called a cut point. *Otherwise, it is called a* simple node.

$$\frac{n \notin H.N \quad H.V(p) = \sharp}{\texttt{post}(p\texttt{=new},H) = \texttt{addNode}(p,n)(H)}\ \text{a-new} \qquad \frac{H' = \texttt{unfold}(p)(H) \quad H'.V(p) = \sharp}{\texttt{post}(p\texttt{->next=null},H) = H_{err}}\ \text{a-ptr1}$$

$$\frac{H' = \texttt{unfold}(p)(H) \quad H'.V(p) \neq \sharp}{\texttt{post}(p\texttt{->next=null},H) = \texttt{delGarbage}(\texttt{upd}S(\texttt{get}V(p),\sharp)(H'))}\ \text{a-ptr2}$$

$$\frac{H.V(p) \neq \sharp \quad \texttt{eval}(dt)(H) \neq \bot}{\texttt{post}(p\texttt{->data=}dt,H) = \texttt{upd}L(\texttt{get}V(p),dt)(H)}\ \text{a-d1} \qquad \frac{H.V(p) = \sharp}{\texttt{post}(p\texttt{->data=}dt,H) = H_{err}}\ \text{a-d2}$$

Fig. 2. A fragment of the definition of $\texttt{post}(St,H)$

In the following, we consider only heaps without garbage, i.e., all the nodes are reachable from nodes labeled by pointer variables. For simplicity, we suppose that each pointer assignment `p->next=q`, resp. `p=q`, is preceded by `p->next=null`, resp. `p=null`. We define a postcondition operator, denoted $\texttt{post}(St,H)$, for any statement St and any heap H. Figure 2 illustrates a part of its definition that contains all the important graph transformations; the full definition is provided in [2]. A collecting semantics can be defined as usual by extending `post` to sets of heaps. The heap H_{err} is a special value denoting the sink heap configuration obtained when `null` dereferences are done.

The formal definition of operators used in this semantics is given on Figure 3. To access the components of a heap H, we use the dotted notation, e.g., $H.N$ denotes the set of nodes of H. For components which are functions, e.g., S, we use currified operators `get` to apply these components to any heap. In the conclusion of the rule a-ptr2, we abuse notation by letting $\sharp$ denote the constant function which associates $\sharp$ to each node. For instance, $\texttt{get}S(f_n)(H)$ returns the successor in H of a node denoted by $f_n(H)$. Similarly, we use the `upd` operators to alter the components of heaps. The operator $\texttt{addNode}(p,n)(H)$ adds a fresh node n (not in $H.N$) to H s.t. it is pointed by p and its data is arbitrary. The `eval` operator evaluates data terms in a concrete heap to integer values or, when `null` is dereferenced, to $\bot$. The operator $\texttt{unfold}(p)(H)$ is introduced to obtain similar definitions for the concrete and abstract program semantics; in the concrete semantics, it is the identity. The operator $\texttt{delGarbage}(H)$ removes from the heap all the garbage nodes using two operators: (1) $\texttt{getGarbage}(H)$ returns the complete set of garbage nodes (computed, e.g., by a graph traversal algorithm starting from the nodes pointed by program variables); (2) $\texttt{proj}(N)(H)$ removes from H a set of nodes $N \subset H.N$ ($f \uparrow N$ denotes a function obtained from f by removing from its domain the set N).

$$\begin{array}{rl}
\texttt{get}V(p)(H) \stackrel{\texttt{def}}{=} H.V(p) \quad \texttt{get}S(f_n)(H) \stackrel{\texttt{def}}{=} H.S(f_n(H)) & \texttt{unfold}(p)(H) \stackrel{\texttt{def}}{=} H \\
\texttt{addNode}(p,n)(H) \stackrel{\texttt{def}}{=} (H.N, H.S[n \mapsto \sharp], H.V[p \mapsto n], H.L[n \mapsto v], H.D) & \text{for some } v \in \mathbf{Z} \\
\texttt{delGarbage}(H) \stackrel{\texttt{def}}{=} \texttt{proj}(\texttt{getGarbage}(H))(H) & \\
\texttt{proj}(N)(H) \stackrel{\texttt{def}}{=} (H.N \setminus N, (H.S \uparrow N)[\sharp/n]_{n \in N}, (H.V)[\sharp/n]_{n \in N}, H.L \uparrow N, H.D) & \\
\texttt{upd}S(f_n,f_m)(H) \stackrel{\texttt{def}}{=} (H.N, H.S[f_n(H) \mapsto f_m(H)], H.V, H.L, H.D) & \\
\texttt{upd}L(f_n,dt)(H) \stackrel{\texttt{def}}{=} (H.N, H.S, H.V, H.L[f_n(H) \mapsto \texttt{eval}(dt)(H)], H.D) &
\end{array}$$

Fig. 3. Operators used in $\texttt{post}(St,H)$

3 Abstract Domain for Program Configurations

Starting from a heap H, we can define a *precise abstraction* by (1) a graph G containing as nodes at least all the cut points in H such that two nodes in G are connected by an edge if there exists a path between them in H, and (2) a constraint $\widetilde{W}$ that associates to each node n in G, let m be its successor, a word over $\mathbb{Z}$ which represents the data values of the path $nn_1 \dots n_k$ from H, where n_k is a predecessor of m. For example, Figures 4(b–c) give precise abstractions for the heap in Figure 4(a). Coarser abstractions can

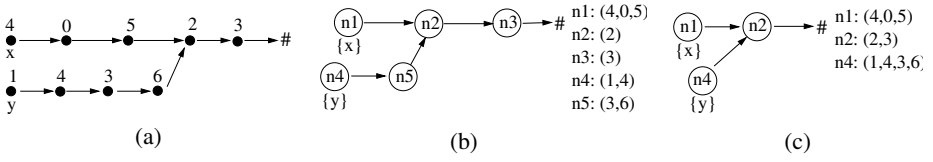
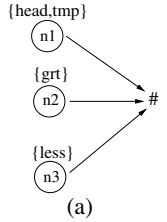


Fig. 4. Concrete and abstract representations for program heaps

be obtained by replacing $\widetilde{W}$ with less precise constraints characterizing the data words attached to the nodes of the graph and the values of the program data variables. For that, abstract domains on words are used which capture various aspects such as constraints on the sizes, the multisets of their elements, or the data at different positions of the words. For example, Figures 5(a–b) give a precise abstraction of the heap configuration at line 2 of the procedure `Dispatch3` from Figure 1(b). Another abstraction of this configuration is defined using the same graph together with the constraints from Figure 5(c) which characterize the data at any position of the words attached to the nodes of the graph. These constraints are expressed by (universally quantified) first-order formulas where $\texttt{hd}(n2)$ denotes the first symbol of the word denoted by $n2$, y is a variable interpreted as a position in some word, $y \in \texttt{tl}(n2)$ means that y belongs to the tail of $n2$, and $n2[y]$ is a term interpreted as the data at the position y of $n2$.



Precise constraints:

$n1 : (3,4,5)$
$n2 : (5,6)$
$n3 : (1,2)$

(b)

Universally quantified formulas:

$\texttt{hd}(n2) \geq 3 \wedge \texttt{hd}(n3) < 3$
$\forall y.\ y \in \texttt{tl}(n2) \Rightarrow n2[y] \geq 3$
$\forall y.\ y \in \texttt{tl}(n3) \Rightarrow n3[y] < 3$

(c)

Fig. 5. Different abstractions for some configuration of `Dispatch3`

3.1 Preliminaries on Abstract Interpretation

Let $\mathcal{C} = (C, \subseteq)$ and $\mathcal{A} = (A, \sqsubseteq)$ be two lattices ($\subseteq$, resp. $\sqsubseteq$, are order relations on C, resp. A). The lattice $\mathcal{A}$ is an *abstract domain* for $\mathcal{C}$ [5] if there exists a Galois connection between $\mathcal{C}$ and $\mathcal{A}$, that is, a pair of monotone functions $(\alpha : C \to A, \gamma : A \to C)$ such that for any $c \in C$ and $a \in A$, $\alpha(c) \sqsubseteq a$ iff $c \subseteq \gamma(a)$. Also, $\mathcal{C}$ is called the *concrete domain* for

$\mathcal{A}$. In the following, an abstract domain $\mathcal{A}$ is denoted by $\mathcal{A} = (A, \sqsubseteq, \sqcap, \sqcup, \top, \bot)$, where $\sqcap$ denotes its greatest lower bound (meet) operator, $\sqcup$ denotes its lowest greater bound (join) operator, $\top$ its top element and $\bot$ its bottom element. Moreover, as usual in the abstract interpretation framework, ∇ represents the widening operator.

Let $\mathcal{F}_C$ be a set of concrete transformers, that is, of functions from C into C. If $\mathcal{A}$ is an abstract domain for $\mathcal{C}$, the set of its abstract transformers, denoted $\mathcal{F}^{\#}_{\mathcal{A}}$, contains a function $f^{\#} : A \to A$ for each $f \in \mathcal{F}_C$. The transformer $f^{\#}$ is *sound* if $f(\gamma(a)) \subseteq \gamma(f^{\#}(a))$, for any $a \in A$. $f^{\#}$ is a *best abstraction* if $\alpha(f(\gamma(a))) = f^{\#}(a)$ and it is an *exact abstraction* if $f(\gamma(a)) = \gamma(f^{\#}(a))$, for any abstract value $a \in A$.

3.2 Data Words (Abstract) Domains

To represent constraints on words associated to the nodes of the graph as in Figure 5(b)-(c) and the values of the program data variables we use elements from a *data words abstract domain*. Let *DWVar* be a set of variables called data word variables and let $\mathbb{Z}^+$ denote the set of non-empty sequences over $\mathbb{Z}$. Also, let $\mathtt{hd}(w)$ (and $\mathtt{tl}(w)$) denote the first element (resp. the tail) of the word w, $[]$ (and $[e]$) the empty word (resp. the word with one element e), and @ the concatenation operator.

Definition 2. *The* data words domain over *DWVar* and *DVar, denoted by* $\mathcal{C}_{\mathbb{W}}(DWVar, DVar)$*, is the lattice of sets of pairs* (L, D) *with* $L : DWVar \to \mathbb{Z}^+$ *and* $D : DVar \to \mathbb{Z}$.

For any data words domain, we define a set of transformers, denoted by $\mathcal{F}_{\mathcal{C}_{\mathbb{W}}}$, as follows ($w, w' \in DWVar$, $d \in DVar$, $W \in \mathcal{C}_{\mathbb{W}}$):

– $\mathtt{addSglt}(w, W)$ adds to each pair (L, D) of W a new word w s.t. $\mathtt{tl}(L(w)) = []$, i.e., w has only one element,

– $\mathtt{selectSglt}(w, W)$ (resp. $\mathtt{selectNonSglt}(w, W)$) selects from W the pairs (L, D) for which $\mathtt{tl}(L(w)) = []$ (resp. $\mathtt{tl}(L(w)) \neq []$), i.e., pairs where the word w has one element (resp. at least two elements),

– $\mathtt{split}(w, w', W)$ changes the L component of each pair in W by adding a new word w' and then assigning $[\mathtt{hd}(L(w))]$ to $L(w)$ and $\mathtt{tl}(L(w))$ to $L(w')$,

– $\mathtt{updFst}(w, dt, W)$ changes the L component of each pair (L, D) of W s.t. $\mathtt{hd}(L(w))$ takes the value of the arithmetic expression dt in which the basic terms are integer constants, data variables, and terms of the form $\mathtt{hd}(w')$ with $w' \in DWVar$,

– $\mathtt{proj}(U, W)$ removes from the domain of L the variables in U, for each $(L, D) \in W$,

– $\mathtt{concat}(V, W)$, where V is a vector of data word variables, changes the L component of each pair (L, D) of W by assigning to $V[0]$ the concatenation of the words represented by the variables in V, i.e., $L(V[0])@\cdots@L(V[|V|-1])$ and by projecting out the variables in V except the first one. Then, $\mathtt{concat}(V_1, \ldots, V_t, W)$ is the component-wise extension of $\mathtt{concat}(V, W)$ to t vectors of data word variables, for any $1 \leq t$.

Definition 3. $\mathcal{A}_{\mathbb{W}} = \left(A^{\mathbb{W}}, \sqsubseteq^{\mathbb{W}}, \sqcap^{\mathbb{W}}, \sqcup^{\mathbb{W}}, \top^{\mathbb{W}}, \bot^{\mathbb{W}}\right)$ *is a* $\mathbb{DW}$-domain over *DWVar* and *DVar if it is an abstract domain for* $\mathcal{C}_{\mathbb{W}}(DWVar, DVar)$*. Let* $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ *denote the set of abstract transformers corresponding to* $\mathcal{F}_{\mathcal{C}_{\mathbb{W}}}$.

We define in Section 4 a $\mathbb{DW}$-domain which formalizes the abstraction from Figure 5(c). Moreover, we define in [2] the $\mathbb{DW}$-domain $\mathcal{A}_{\Sigma}$ (resp. $\mathcal{A}_{\mathbb{M}}$) representing constraints over the sum (resp. the multiset) of data in a word.

3.3 The Domain of Abstract Heap Sets

In the following, we assume that for each node of a heap there exists a data word variable with the same name.

Definition 4. *An* abstract heap over *PVar*, *DVar*, and a $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{W}}$ *is a tuple* $\widetilde{H} = \langle N, S, V, \widetilde{W} \rangle$ *where* N, S, V *are as in the definition of heaps, and* $\widetilde{W}$ *is an abstract value in* $\mathcal{A}_{\mathbb{W}}$ *over the data word variables* $N \setminus \{\sharp\}$ *and the data variables DVar. A* k-abstract heap *is an abstract heap with at most* k *simple nodes.*

An example of an abstract heap is given in Figure 5(a) and (c). Two abstract heaps are *isomorphic* if their underlying graphs are isomorphic. Let $\mathcal{C}_{\mathbb{H}}$ denote the lattice of sets of heaps. We define $\mathcal{A}_{\mathbb{H}}(k, \mathcal{A}_{\mathbb{W}})$ an abstract domain for $\mathcal{C}_{\mathbb{H}}$ whose elements are k-abstract heaps over $\mathcal{A}_{\mathbb{W}}$ s.t. (1) for any two isomorphic abstract heaps, the lattice operators are obtained by applying the corresponding operators between the values from $\mathcal{A}_{\mathbb{W}}$, and (2) the join and the widening (resp. meet) of two non-isomorphic abstract heaps is $\top^{\mathbb{H}}$ (resp. $\bot^{\mathbb{H}}$). Notice that $\nabla^{\mathbb{H}}$ is a widening operator because the heaps generated by the programs we consider (see Section 2) contain a bounded number of cut points [15].

Finally, we define $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}}) = \left(A^{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}}), \sqsubseteq^{\mathbb{HS}}, \sqcap^{\mathbb{HS}}, \sqcup^{\mathbb{HS}}, \top^{\mathbb{HS}}, \bot^{\mathbb{HS}}\right)$ as a finite powerset domain over $\mathcal{A}_{\mathbb{H}}(k, \mathcal{A}_{\mathbb{W}})$. Its elements are called *k-abstract heap sets*. Obviously, $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}})$ is an abstract domain for $\mathcal{C}_{\mathbb{H}}$.

Definition 5. *A* k-abstract heap set over *PVar*, *DVar*, and a $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{W}}$ *is a finite set of non-isomorphic k-abstract heaps over PVar, DVar, and* $\mathcal{A}_{\mathbb{W}}$.

The operators associated to $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}})$ and its widening operator are obtained from those of $\mathcal{A}_{\mathbb{H}}(k, \mathcal{A}_{\mathbb{W}})$ as usual [6]. For example, the join of two abstract heap sets is computed by taking the union of these sets and by applying the join operator between any two isomorphic abstract heap graphs.

3.4 Abstract Postcondition Transfomer

The abstract postcondition transformer $\texttt{post}^{\#}$ on abstract heap sets is obtained by replacing in the definition of $\texttt{post}$ (see Figure 2) the heap H by an abstract heap set A_H and every concrete operator defined in Figure 3 by its abstract version given in Figure 6.

Next, $\texttt{post}^{\#}$ is extended to obtain a postcondition transformer on k-abstract heap sets, denoted $\texttt{post}^{\#}_k$, by

$$\texttt{post}^{\#}_k(St, A_H) \stackrel{\texttt{def}}{=} \bigsqcup^{\mathbb{HS}}_{\widetilde{H} \in \texttt{post}^{\#}(St, A_H)} \texttt{Normalize}^{\#}_k(\widetilde{H}),$$

where $\texttt{Normalize}^{\#}_k$ takes as input an abstract heap graph $\widetilde{H}$ and, if $\widetilde{H}$ is not a k-abstract heap graph then it returns a 0-abstract heap graph (an abstract heap graph with no simple nodes). Suppose that $V_1, \ldots, V_t$ are all the (disjoint) paths in $\widetilde{H}$ of the form $nn_1 \ldots n_k$, where $k \geq 1$, n is a cut point, n_i is a simple node, for any $1 \leq i \leq k$, and the successor of n_k is a cut point. $\texttt{Normalize}^{\#}_k$ calls the transformer $\texttt{concat}^{\#}(V_1, \ldots, V_t, \widetilde{W})$, then it replaces the paths of simple nodes starting from $V_i[0]$ by one edge, and finally it removes from the graph the simple nodes of each V_i.

Remark 1. By definition 4, for any call to $\texttt{concat}^{\#}(V_1, \ldots, V_t, \widetilde{W})$ made by $\texttt{post}^{\#}_k$, the sum $|V_1| + \ldots + |V_t|$ is bounded by $2k$. Consequently, we can define transformers $\texttt{concat}^{\#}$ s.t. $\texttt{concat}^{\#}(V_1, \ldots, V_t, \widetilde{W})$ with $\widetilde{W} \in \mathcal{A}_{\mathbb{W}}$ is undefined if $|V_1| + \ldots + |V_t| > 2k$.

$$F(A_H) \stackrel{\texttt{def}}{=} \bigsqcup^{\mathbb{HS}}_{\widetilde{H}\in A_H} F(\widetilde{H}),\ \text{for any } F \in \{\texttt{get}^{\#}V(p), \texttt{get}^{\#}S(f_n), \texttt{unfold}^{\#}(n), \texttt{addNode}^{\#}(p,n), \texttt{delGarbage}^{\#}, \texttt{proj}^{\#}(N), \texttt{upd}S^{\#}(f_n,f_m), \texttt{upd}L^{\#}(f_n,dt)\}$$

$$\begin{array}{rcl}
\texttt{get}^{\#}V(p)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \widetilde{H}.V(p) \quad \texttt{get}^{\#}S(f_n)(\widetilde{H}) \stackrel{\texttt{def}}{=} \widetilde{H}.S(f_n(\widetilde{H})) \\
\texttt{unfold}^{\#}(n)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \langle \widetilde{H}.N, \widetilde{H}.S, \widetilde{H}.V, \texttt{selectSglt}^{\#}(n, \widetilde{H}.\widetilde{W})\rangle \\
& \sqcup^{\mathbb{HS}} & \langle \widetilde{H}.N \cup \{m\}, \widetilde{H}.S[n \mapsto m, m \mapsto \widetilde{H}.S(n)], \widetilde{H}.V, \\
& & \quad \texttt{selectSglt}^{\#}(n, \texttt{split}^{\#}(n,m, \texttt{selectNonSglt}^{\#}(n, \widetilde{H}.\widetilde{W})))\rangle \\
\texttt{addNode}^{\#}(p,n)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \langle \widetilde{H}.N, \widetilde{H}.S[n \mapsto \sharp], \widetilde{H}.V[p \mapsto n], \texttt{addSglt}^{\#}(n, \widetilde{H}.\widetilde{W})\rangle \\
\texttt{delGarbage}^{\#}(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \texttt{proj}^{\#}(\texttt{getGarbage}(\widetilde{H}))(\widetilde{H}) \\
\texttt{proj}^{\#}(N)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \langle \widetilde{H}.N \setminus N, (\widetilde{H}.S \upharpoonright N)[\sharp/n]_{n\in N}, (\widetilde{H}.V)[\sharp/n]_{n\in N}, \texttt{proj}^{\#}(N, \widetilde{H}.\widetilde{W})\rangle \\
\texttt{upd}S^{\#}(f_n,f_m)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \langle \widetilde{H}.N, \widetilde{H}.S[f_n(\widetilde{H}) \mapsto f_m(\widetilde{H})], \widetilde{H}.V, \widetilde{H}.\widetilde{W}\rangle \\
\texttt{upd}L^{\#}(f_n,dt)(\widetilde{H}) & \stackrel{\texttt{def}}{=} & \langle \widetilde{H}.N, \widetilde{H}.S, \widetilde{H}.V, \texttt{updFst}^{\#}(f_n(\widetilde{H}), dt[\texttt{hd}(\widetilde{H}.V(p))/p\texttt{->data}]_{p\in PVars}, \widetilde{H}.\widetilde{W})\rangle
\end{array}$$

Fig. 6. Operators used in $\texttt{post}^{\#}(St, A_H)$

Theorem 1. *For any k-abstract heap set A_H in $\mathcal{A}_{\mathbb{HS}}(k, \mathcal{A}_{\mathbb{W}})$, the following hold:*
– (soundness) $\texttt{post}(St, \gamma^{\mathbb{HS}}(A_H)) \sqsubseteq^{\mathbb{HS}} \gamma^{\mathbb{HS}}(\texttt{post}^{\#}_k(St, A_H))$*;*
– (precision) if all the abstract transformers in $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ of the domain $\mathcal{A}_{\mathbb{W}}$ are best (exact, resp.) abstractions then $\texttt{post}^{\#}_k$ *is also a best (exact, resp.) abstraction.*

We define in the next section a $\mathbb{DW}$-domain for which $\mathcal{F}^{\#}_{\mathcal{A}_{\mathbb{W}}}$ contains sound and best abstract transformers. [2] presents other sound transformers for $\mathbb{DW}$-domains representing sum and multiset constraints.

4 A $\mathbb{DW}$-Domain with Universally Quantified Formulas

We define the $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{U}} = \left(A^{\mathbb{U}}, \sqsubseteq^{\mathbb{U}}, \sqcap^{\mathbb{U}}, \sqcup^{\mathbb{U}}, \top^{\mathbb{U}}, \bot^{\mathbb{U}}\right)$ whose elements are first-order formulas with free variables in $DWVar \cup DVar$.

4.1 Abstract Domain Definition

The formulas in $A^{\mathbb{U}}$ contain a quantifier-free part and a conjunction of universally quantified formulas of the form $\forall \mathbf{y}.\ (P \Rightarrow U)$, where $\mathbf{y}$ is a vector of (integer) variables representing positions in the words, the *guard* P is a constraint over $\mathbf{y}$, and U is a quantifier-free formula over $DVar$, $\mathbf{y}$, and $DWVar$. The formula P is defined using a finite set $\mathcal{P}$ of *guard patterns* considered as a parameter of $\mathcal{A}_{\mathbb{U}}$.

Syntax of guard patterns: Let $\mathbb{O} \subseteq DWVar$ be a set of distinguished data word variables and $\omega_1, \ldots, \omega_n \in \mathbb{O}$. Let $\mathbf{y}_1, \ldots, \mathbf{y}_n$ be non-empty vectors of *position variables* interpreted as positions in the words denoted by $\omega_1, \ldots, \omega_n$ (these variables are universally quantified in the elements of $A^{\mathbb{U}}$). We assume that these vectors are pairwise disjoint and that $\omega_i \neq \omega_j$, for any $i \neq j$. We denote by y_i^j the jth element of the vector $\mathbf{y}_i$, $1 \leq j \leq |\mathbf{y}_i|$. Let $\Omega \subseteq \mathbb{O}$ be a set of variables not necessarily distinct from $\omega_1, \ldots, \omega_n$.

The guard patterns are conjunctions of (1) a formula that associates vectors of position variables with data word variables, (2) an arithmetical constraint on the values of

some position variables, and (3) order constraints between the position variables associated with the same data word variable. Formally,

$$P(\mathbf{y}_1,\ldots,\mathbf{y}_n,\omega_1,\ldots,\omega_n,\Omega) ::= \bigwedge_{1\le i\le n} \mathbf{y}_i \in \mathtt{tl}(\omega_i) \wedge P_L(y_1^1,\ldots,y_n^1,\Omega) \wedge \bigwedge_{1\le i\le n} P_R^i(\mathbf{y}_i)$$
$$P_R(y^1y^2\ldots y^m) ::= y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m$$

where (1) for each vector $\mathbf{y}_i$, $\mathbf{y}_i \in \mathtt{tl}(\omega_i)$ states that the positions denoted by $\mathbf{y}_i$ belong to the tail of the word denoted by ω_i and that $\mathtt{len}(\omega_i) \ge |\mathbf{y_i}| + 1$; the terms $\mathtt{len}(\omega_i)$ and $\mathtt{tl}(\omega_i)$ denote the length and the tail of the word represented by ω_i, (2) P_L is a boolean combination of linear constraints over y_i^1 with $1 \le i \le n$; these constraints may use the terms $\mathtt{len}(\omega)$ with $\omega \in \Omega$ but we assume that P_L is not a constraint for the lengths of the words in Ω, i.e., $\wedge_{\omega\in\Omega}\mathtt{len}(\omega) > 0$ implies $\exists y_1^1,\ldots,y_n^1$. P_L (in Presburger arithmetic), (3) for each vector $\mathbf{y}_i$, the formula $P_R(\mathbf{y}_i)$ is an order constraint over $\mathbf{y}_i$, where $\prec_i \in \{\le, <, <_1\}$ with $x <_1 y$ iff $y = x+1$.

The elements of $A^{\mathbb{U}}$: Let $\mathcal{V} \subseteq DWVar$ and let $\mathcal{P}$ be a set of guard patterns. We define $\mathcal{P}(\mathcal{V})$ to be the set of all formulas obtained from some $P(\mathbf{y}_1,\ldots,\mathbf{y}_n,\omega_1,\ldots,\omega_n,\Omega) \in \mathcal{P}$ by substituting each ω_i with some $w_i \in DWVar$, for any $1 \le i \le n$, and Ω with some $\mathbf{W} \subseteq DWVar$. We assume that $w_i \ne w_j$, for any $i \ne j$. Then, an element of $A^{\mathbb{U}}$ has the following syntax:

$$\widetilde{W}(\mathcal{V}) ::= E(\mathcal{V}) \wedge \bigwedge_{g(\mathbf{y})\in\mathcal{P}(\mathcal{V})} \forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U_g \tag{G}$$

where $E(\mathcal{V})$ is a quantifier-free arithmetical formula over $DVar$ and terms $\mathtt{hd}(w)$, $\mathtt{len}(w)$ with $w \in \mathcal{V}$, $g(\mathbf{y})$ is a guard of the form $P(\mathbf{y}_1,\ldots,\mathbf{y}_n,w_1,\ldots,w_n,\mathbf{W})$ with $\mathbf{y} = \mathbf{y}_1 \cup \ldots \cup \mathbf{y}_n$, and U_g is a quantifier-free arithmetical formula over the terms in $E(\mathcal{V})$ together with $w[y]$ and y, for any $w \in DWVar$ and $y \in \mathbf{y}$. The terms $\mathtt{hd}(w)$, resp. $w[y]$, denote the data at the first position, resp. the position denoted by y, of the word represented by w. We assume that E and U_P are also elements of some numerical abstract domain $\mathcal{A}_{\mathbb{Z}} = \left(A^{\mathbb{Z}}, \sqsubseteq^{\mathbb{Z}}, \sqcap^{\mathbb{Z}}, \sqcup^{\mathbb{Z}}, \top^{\mathbb{Z}}, \bot^{\mathbb{Z}}\right)$ which is a parameter of $\mathcal{A}_{\mathbb{U}}$.

Examples: The following formula is an element of $A^{\mathbb{U}}$ parametrized by $\mathcal{P} = \{y_1 \in \mathtt{tl}(\omega_1) \wedge y_2 \in \mathtt{tl}(\omega_2) \wedge y_1 = y_2\}$ and the Polyhedra domain [7]. It expresses the fact that the word denoted by w_1 is a copy of the word denoted by w_2:

$$\begin{array}{c} \mathtt{len}(w_1) = \mathtt{len}(w_2) \wedge \mathtt{hd}(w_1) = \mathtt{hd}(w_2) \\ \wedge \forall y_1, y_2.\, \big((y_1 \in \mathtt{tl}(w_1) \wedge y_2 \in \mathtt{tl}(w_2) \wedge y_1 = y_2) \Rightarrow w_1[y_1] = w_2[y_2]\big) \end{array} \tag{H}$$

The following element of $A^{\mathbb{U}}$ over $\mathcal{P} = \{(y_1,y_2,y_3) \in \mathtt{tl}(\omega) \wedge y_1 <_1 y_2 <_1 y_3\}$ and the Polyhedra domain represents words w whose data are in the Fibonacci sequence:

$$\mathtt{hd}(w) = 1 \wedge \forall y_1,y_2,y_3.\, \big((y_1,y_2,y_3) \in \mathtt{tl}(w) \wedge y_1 <_1 y_2 <_1 y_3) \Rightarrow w[y_3] = w[y_1] + w[y_2]\big).$$

Lattice operators: The concretization function for elements in $A^{\mathbb{U}}$ is defined according to the classical semantics of these formulas. The value $\top^{\mathbb{U}}$ (resp. $\bot^{\mathbb{U}}$) is defined by the formula in which E and all U_g are $\top^{\mathbb{Z}}$ (resp. $\bot^{\mathbb{Z}}$). Let

$$\widetilde{W}(\mathcal{V}_1) = E(\mathcal{V}_1) \wedge \bigwedge_{g(\mathbf{y})\in\mathcal{P}(\mathcal{V}_1)} \forall \mathbf{y}.\, (g(\mathbf{y}) \Rightarrow U_g) \text{ and } \widetilde{W}'(\mathcal{V}_2) = E'(\mathcal{V}_2) \wedge \bigwedge_{g(\mathbf{y})\in\mathcal{P}(\mathcal{V}_2)} \forall \mathbf{y}.\, (g(\mathbf{y}) \Rightarrow U'_g).$$

Before applying any lattice operator we add to $\widetilde{W}$ (resp. $\widetilde{W}'$) universally quantified formulas $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow \top^{\mathbb{Z}}$, for any $g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \setminus \mathcal{P}(\mathcal{V}_2)$ (resp. $g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_2) \setminus \mathcal{P}(\mathcal{V}_1)$). Then, $\widetilde{W} \sqsubseteq^{\mathbb{U}} \widetilde{W}'$ iff (1) $E \sqsubseteq^{\mathbb{Z}} E'$, and (2) for each $g(\mathbf{y}) = P(\mathbf{y}_1, \ldots, \mathbf{y}_n, w_1, \ldots, w_n, \mathbf{W}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)$, if for all $i \in [1,n]$, $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(w_i) \geq |\mathbf{y}_i| + 1$ then $E \sqcap^{\mathbb{Z}} U_g \sqsubseteq^{\mathbb{Z}} U'_g$. Also, $\widetilde{W} \sqcup^{\mathbb{U}} \widetilde{W}'$ is defined by $E \sqcup^{\mathbb{Z}} E' \wedge \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}_1) \cup \mathcal{P}(\mathcal{V}_2)} \forall \mathbf{y}.\ (g(\mathbf{y}) \Rightarrow U_g \sqcup^{\mathbb{Z}} U'_g)$. The operators $\sqcap^{\mathbb{U}}$ and $\nabla^{\mathbb{U}}$ are defined in a similar way.

In the following, we present the two most interesting abstract transformers in $\mathcal{F}^{\#}_{\mathcal{A}^{\mathbb{U}}}$, $\mathtt{split}^{\#}$ and $\mathtt{concat}^{\#}$. For the sake of readability, we present these transformers only for guard patterns of the form $P(\mathbf{y}_1, \omega_1, \Omega)$, i.e., patterns with positions belonging to only one word. The general case is given in [2]. At the end of this section, we give sufficient conditions to obtain soundness and precision for all transformers in $\mathcal{F}^{\#}_{A^{\mathbb{U}}}$.

4.2 Abstract Transformer $\mathtt{split}^{\#}$

Let $\widetilde{W}(\mathcal{V}) = E(\mathcal{V}) \wedge \phi(\mathcal{V}) \in A^{\mathbb{U}}$ as in (G). The transformer $\mathtt{split}^{\#}(u, v, \widetilde{W})$ splits u into its head and its tail; the head is assigned to u and the tail to v. It produces a formula $E'(\mathcal{V} \cup \{v\}) \wedge \phi'(\mathcal{V} \cup \{v\})$, where $\phi'(\mathcal{V} \cup \{v\}) = \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V} \cup \{v\})} \forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$ and:

1. E' is obtained from E by: (1) adding constraints on $\mathtt{hd}(v)$ obtained from $\phi(\mathcal{V})$, (2) substituting $\mathtt{len}(u)$ with $\mathtt{len}(v) + 1$ and assigning 1 to $\mathtt{len}(u)$,
2. ϕ' is obtained from ϕ by: (1) adding constraints on $\mathtt{tl}(v)$ computed from the constraints on $\mathtt{tl}(u)$ in ϕ, and then, (2) applying the second step from the computation of E' to the right hand side of each implication.

In the following, we detail only the important steps.

Constraints on $\mathtt{hd}(v)$: Let $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U_g$ be a conjunct of ϕ with $\mathbf{y} = y^1 \ldots y^m$ and $g(\mathbf{y}) = \mathbf{y} \in \mathtt{tl}(u) \wedge y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m \wedge P_L(y^1, \mathbf{W})$. A constraint on $\mathtt{hd}(v)$ in $\widetilde{W}'$ is deduced from U_g when y^1 coincides with the first position in v. Formally, if the Presburger formula $y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 = 1$ is satisfiable then

$$E'_g = E \sqcap^{\mathbb{Z}} \left(U_g \uparrow^{\mathbb{Z}} (\mathbf{y} \setminus y^1)\right) \left[1/y^1, \mathtt{hd}(v)/u[y^1]\right]$$

is a constraint on $\mathtt{hd}(v)$. Then, E' is the meet ($\sqcap^{\mathbb{Z}}$) between all abstract values E'_g, computed as above, for every conjunct $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U_g$ of ϕ with $g(\mathbf{y}) \in \mathcal{P}(\{u\})$.

Constraints on $\mathtt{tl}(v)$: The formulas that constrain the tail of v are of the form $\forall \mathbf{y}.\ g(\mathbf{y}) \Rightarrow U'_g$, where $g(\mathbf{y}) \in \mathcal{P}(\{v\})$. We compute simultaneously all U'_g with $g(\mathbf{y}) \in \mathcal{P}(\{v\})$ as follows: (1) we start with $U'_g = \top^{\mathbb{Z}}$, for every $g(\mathbf{y}) \in \mathcal{P}(\{v\})$, and (2) for every $g(\mathbf{y}) = \mathbf{y} \in \mathtt{tl}(u) \wedge y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m \wedge P_L(y^1, \mathbf{W})$ in $\mathcal{P}(\mathcal{V})$, we do the following:

- if $y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 > 1$ is satisfiable, then let $g' = g[v/u]$ and $U'_{g'} = U'_{g'} \sqcap^{\mathbb{Z}} U_g$;
- if $y^1 \prec_1 y^2 \prec_2 \ldots \prec_m y^m \wedge P_L(y^1, \mathbf{W}) \wedge E \wedge y^1 = 1$ is satisfiable and $|\mathbf{y}| > 1$, then we try to generate a universal formula constraining $|\mathbf{y}| - 1$ positions that holds on the tail of v. Let $\mathbf{y}' = y^2 \ldots y^m$ and $g'(\mathbf{y}') = \mathbf{y}' \in \mathtt{tl}(v) \wedge y^2 \prec_2 \ldots \prec_m y^m$. If $g'(\mathbf{y}') \in \mathcal{P}(\{v\})$ then $U'_{g'} = U'_{g'} \sqcap^{\mathbb{Z}} U_g \uparrow^{\mathbb{Z}} (\{y^2, \ldots, y^m\})$.

4.3 Abstract Transformer $\texttt{concat}^{\#}$

Let $\widetilde{W}(\mathcal{V}) = E(\mathcal{V}) \wedge \phi(\mathcal{V}) \in A^{\mathbb{U}}$ and $V = v_1 \ldots v_n$ be a vector of variables in $\mathcal{V}$. The transformer $\texttt{concat}^{\#}(V, \widetilde{W})$ assigns to v_1 the concatenation of the words represented by the variables from V in $\widetilde{W}$ and it removes $v_2, \ldots, v_n$.

Let $\alpha_1\beta_1 \ldots \alpha_m\beta_m\alpha_{m+1}$ be a decomposition of V, where β_i are maximal sub-vectors of V of length at least 2 such that for any $1 \leq i \leq m$ and $v \in \beta_i$, $E \sqsubseteq^{\mathbb{Z}} \texttt{len}(v) = 1$ (α_1 and α_{m+1} may be empty). We define $\widetilde{W}'(\mathcal{V}') = \texttt{concat}^{\#}(V, \widetilde{W}(\mathcal{V}))$ in two steps:

1. we concatenate the singleton words of each β_i, for $i = 1$ to n. Let $\widetilde{W}_0(\mathcal{V}'_0) = \widetilde{W}(\mathcal{V})$ where $\mathcal{V}'_0 = \mathcal{V}$, and for every $1 \leq i \leq m$, let $\widetilde{W}_{i+1}(\mathcal{V}'_{i+1}) = \texttt{concat}^{\#}(\beta_i, \widetilde{W}_i(\mathcal{V}'_i))$ where $\mathcal{V}'_{i+1} = \mathcal{V}'_i \setminus \texttt{tl}(\beta_i)$. This step generates universally quantified formulas on $\texttt{hd}(\beta_i)$ by collecting from E the constraints on $\texttt{hd}(v)$ with $v \in \texttt{tl}(\beta_i)$;
2. let $\alpha' = \alpha_1\texttt{hd}(\beta_1)\ldots\alpha_m\texttt{hd}(\beta_m)\alpha_{m+1}$. Notice that α' does not contain two successive variables denoting singletons. We define $\widetilde{W}'(\mathcal{V}') = \texttt{concat}^{\#}(V', \widetilde{W}_m(\mathcal{V}'_m))$, where $\mathcal{V}' = \mathcal{V}'_m \setminus \texttt{tl}(\alpha')$.

The result of $\texttt{concat}^{\#}(V, \widetilde{W}(\mathcal{V}))$ is a formula of the form $E'(\mathcal{V}') \wedge \phi'(\mathcal{V}')$, where $\phi'(\mathcal{V}') = \bigwedge_{g(\mathbf{y}) \in \mathcal{P}(\mathcal{V}')} \forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U'_g$ and:

- E' is obtained from E by: (1) updating the length constraints, i.e. $\texttt{len}(v_1) = \texttt{len}(v_1) + \ldots + \texttt{len}(v_n)$, and (2) projecting out $\texttt{hd}(v)$ and $\texttt{len}(v)$ with $v \in V \setminus \{v_1\}$,
- ϕ' is obtained from ϕ and E by replacing each sub-formula $\forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U_g$ of ϕ with $\forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U'_g$ such that:
  1. if $g(\mathbf{y}) \notin \mathcal{P}(V)$ (g does not constrain the words denoted by V), then U'_g is obtained from U_g by applying the same transformations as for E';
  2. if $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$ then U'_g is the strongest possible constraint that we can compute from $\widetilde{W}$ which characterizes the data from the tail of v_1, knowing that v_1 represents in $\widetilde{W}'$ the concatenation of the words denoted by $v_1, \ldots, v_n$ in $\widetilde{W}$.

We give hereafter the computation of the sub-formulas of ϕ' over v_1 when V is a sequence of singletons, and when V doesn't contain more than two successive singletons.

Concatenating words of length one: Let $V = v_1 \ldots v_n$ be a vector of data word variables in $\mathcal{V}$ such that $E \sqsubseteq^{\mathbb{Z}} \texttt{len}(v_i) = 1$, for all $1 \leq i \leq n$.

Let $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$ be a guard with $|\mathbf{y}| \leq n$. For every $\sigma_g : \mathbf{y} \to [2..n]$, if $g(\mathbf{y})\ \sigma_g \wedge E$ is a satisfiable Presburger formula, then let E_{σ_g} be the numerical abstract value computed from E by (1) substituting, for any i in the co-domain of σ_g, each occurrence of $\texttt{hd}(v_i)$ with $v_1[\sigma_g^{-1}(i)]$, (2) substituting each $\texttt{len}(v_i)$ with 1, and (3) projecting out all the terms $\texttt{hd}(v_i)$ with i not in the co-domain of σ_g. We define U'_g as the join of all abstract values E_{σ_g} computed as above.

Example 1. Suppose that we analyse the procedure `Dispatch3` from Figure 1 using $\mathcal{A}_{\mathbb{HS}}(1, \mathcal{A}_{\mathbb{U}})$ where $\mathcal{A}_{\mathbb{U}}$ is parametrized by $\mathcal{P} = \{y \in \texttt{tl}(\omega_1)\}$ and the Polyhedra domain. Also, suppose that the initial abstract configuration is the one from Figure 7(a). After several iterations of the loop, one of the obtained abstract heaps is pictured in Figure 7(c). It is obtained by applying $\texttt{Normalize}^{\#}_k$ on the abstract heap in Figure 7(b), which calls $\texttt{concat}^{\#}(n_2, n_1, n, \widetilde{W})$ where $\widetilde{W}$ is the formula in Figure 7(b). To compute U such

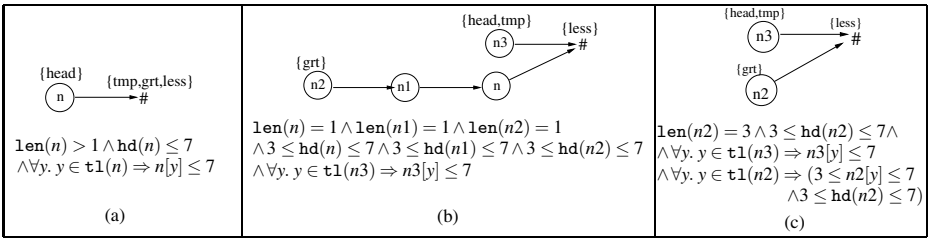


Fig. 7. Abstract heaps for the procedure `Dispatch3`

that $\forall y.\, y \in \mathtt{tl}(n2) \Rightarrow U$ is a sub-formula of $\widetilde{W}'$ we use $\sigma_1(y) = 1$ and $\sigma_2(y) = 2$ and we define $E_{\sigma_i} = 3 \leq n2[y] \leq 7 \wedge 3 \leq \mathtt{hd}(n2) \leq 7$, for all $1 \leq i \leq 2$. Then, $U = E_{\sigma_1} \sqcup^{\mathbb{Z}} E_{\sigma_2}$.

Concatenating words of length greater than one: Let $V = v_1, \ldots, v_n$ be a vector of variables from $\mathcal{V}$ s.t. there exists no $1 \leq i < n$ with $E \sqsubseteq^{\mathbb{Z}} \mathtt{len}(v_i) = 1 \wedge \mathtt{len}(v_{i+1}) = 1$.

Our aim is to generate, for every $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$, a formula of the form $\forall \mathbf{y}.\, g(\mathbf{y}) \Rightarrow U'_g$ on the concatenation of $v_1,\ldots,v_n$ from formulas describing in $\widetilde{W}$ properties of each of the v_is. In order to obtain non-trivial properties (i.e., U'_g is not simply $\top^{\mathbb{Z}}$) we need that the set of guard patterns $\mathcal{P}$ contains "enough" patterns to capture relations on elements within the input words $v_1,\ldots,v_n$. This is ensured by considering a set of patterns denoted by $\mathtt{Closure}(P(\mathbf{y},\omega,\Omega),\omega_1\ldots,\omega_n)$, where $P(\mathbf{y},\omega,\Omega)$ is the pattern used to define the guard $g(\mathbf{y})$.

We give here a brief description of $\mathtt{Closure}(P(\mathbf{y},\omega,\Omega),\omega_1\ldots,\omega_n)$ (the full definition is given in [2]). Assume that ω represents the concatenation of n words denoted by $\omega_1,\ldots,\omega_n \in \mathbb{O}$. Let $\mathbf{p} : \mathbf{y} \to \mathbb{Z}$ be a valuation for the variables $\mathbf{y}$ that satisfies P and $p = \mathbf{p}(y)$, for some $y \in \mathbf{y}$. If $p < \mathtt{len}(\omega_1)$ then p is also a position of the word denoted by ω_1, if $\mathtt{len}(\omega_1) \leq p < \mathtt{len}(\omega_1) + \mathtt{len}(\omega_2)$ then $p - \mathtt{len}(\omega_1)$ is a position of ω_2, etc. In general, with any such p we can associate a position on one of the words $\omega_1,\ldots,\omega_n$. Therefore, for any valuation $\mathbf{p}$ as above we define:

1. a mapping $\sigma : \mathbf{y} \to \{\mathtt{hd}(\omega_1), \mathtt{tl}(\omega_1)\ldots, \mathtt{hd}(\omega_n), \mathtt{tl}(\omega_n)\}$ s.t for any $y \in \mathbf{y}$, $\sigma(y) = \mathtt{hd}(\omega_i)$ iff $\mathbf{p}(y)$ corresponds to the first position of ω_i and $\sigma(y) = \mathtt{tl}(\omega_i)$ iff $\mathbf{p}(y)$ corresponds to a position in the tail of ω_i,
2. n valuations $\mathbf{p}_i : \sigma^{-1}(\mathtt{tl}(\omega_i)) \to \mathbb{Z}$, for any $1 \leq i \leq n$, where $\sigma^{-1}(\mathtt{tl}(\omega_i))$ is the set of variables from $\mathbf{y}$ which are mapped to $\mathtt{tl}(\omega_i)$ by σ and $\mathbf{p}_i(y)$ is the position in the tail of ω_i which corresponds to $\mathbf{p}(y)$.

Let Σ_P be the set of all mappings σ as above. Then, for every $\sigma \in \Sigma_P$, we define a set of patterns $\mathcal{T}_\sigma$ of the form $P'(\sigma^{-1}(\mathtt{tl}(\omega_i)),\omega_i,\Omega)$ with $1 \leq i \leq n$. The pattern $P'(\sigma^{-1}(\mathtt{tl}(\omega_i)),\omega_i,\Omega)$ characterizes the valuations $\mathbf{p}_i$, for any $\mathbf{p}$ a valuation for $\mathbf{y}$ corresponding to σ that satisfies P. Finally, we define $\mathtt{Closure}(P(\mathbf{y},\omega,\Omega),\omega_1\ldots,\omega_n)$ as the union of all the $\mathcal{T}_\sigma$ s.t. $\sigma \in \Sigma_P$. Let $\mathtt{Closure}(P,k)$ denote the union of all $\mathtt{Closure}(P,\omega_1,\ldots,\omega_n)$ with $n \leq 2k$. By Remark 1, $\mathtt{Closure}(P,k)$ is sufficient to handle any call to $\mathtt{concat}^\#$ made by the domain of k-abstract heap sets.

We now have all the ingredients to define the value U'_g with $g(\mathbf{y}) \in \mathcal{P}(\{v_1\})$: (1) assume that $g(\mathbf{y})$ is obtained from $P(\mathbf{y},\omega,\Omega) \in \mathcal{P}$ by some substitution $\gamma : \mathbb{O} \to DWVar$,

(2) let $\sigma \in \Sigma_P$, $\mathcal{T}_\sigma \in \texttt{Closure}(P(\mathbf{y},\omega,\Omega),\omega_1\ldots,\omega_n)$, and $\overline{\mathcal{T}}_\sigma$ be the guards obtained from $\mathcal{T}_\sigma$ by applying γ and by substituting every ω_i with v_i, for any $1 \leq i \leq n$; we define

$$U_{\mathcal{T}_\sigma} = \bigsqcap^{\mathbb{Z}}_{g' \in \overline{\mathcal{T}}_\sigma} U_{g'} \sqcap^{\mathbb{Z}} (E\ \theta)\uparrow\{\texttt{hd}(v), \texttt{len}(v) | v \in \{v_2,\ldots,v_n\}\},$$

where $U_{g'}$ is implied by g' in ϕ and θ substitutes every $\texttt{hd}(v_i)$ for which $\sigma(y) = \texttt{hd}(\omega_i)$, for some $y \in \mathbf{y}$, with $v_1[y]$, and (3) U'_P is the join of all $U'_{\mathcal{T}_\sigma}$ with $\sigma \in \Sigma_P$.

Example 2. Suppose that we analyse the procedure `Fibonacci` from Section 1 using the domain of 3-abstract heap sets parametrized by $\mathcal{A}^{\mathbb{U}}$ over $\mathcal{P} = \texttt{Closure}(P,3)$, where $P((y_1,y_2,y_3),\omega) = (y_1,y_2,y_3) \in \texttt{tl}(\omega) \wedge y_1 <_1 y_2 <_1 y_3$, and the Polyhedra domain.

The analysis starts from an initial state in which `head` points to a non-empty list. After some iterations of the loop, we obtain an abstract heap having 7 nodes in a row, n_i, $1 \leq i \leq 6$, and # such that n_1 and n_6 are pointed by the program variables `head` and `x`, resp. We apply $\texttt{Normalize}^\#_k$ which calls $\texttt{concat}^\#((n_1,n_2,n_3,n_4,n_5),\widetilde{W}_0)$, where $\widetilde{W}_0$ is the formula in $\mathcal{A}^{\mathbb{U}}$ associated to this abstract heap. $\widetilde{W}_0$ is a conjunction between

$$E ::= \texttt{len}(n_1) = 5 \wedge \texttt{hd}(n_1) = 1 \wedge \texttt{hd}(n_2) = 8 \wedge \texttt{hd}(n_3) = 13 \wedge \texttt{hd}(n_4) = 21 \\ \wedge \texttt{hd}(n_5) = 34 \wedge m1 = 13 \wedge m2 = 21 \wedge \textstyle\bigwedge_{2\leq i\leq 5} \texttt{len}(n_i) = 1$$

and some universally-quantified formulas, including

$$\psi ::= \forall y_1,y_2,y_3.\ ((y_1,y_2,y_3) \in \texttt{tl}(n_1) \wedge y_1 <_1 y_2 <_1 y_3) \Rightarrow (n_1[y_3] = n_1[y_1] + n_1[y_2]) \\ \wedge \forall y_1.\ ((y_1) \in \texttt{tl}(n_1) \wedge y_1 = \texttt{len}(n_1) - 1) \Rightarrow (n_1[y_1] = 3),$$

We identify $\alpha_1 = n_1$ and $\beta_1 = (n_2,n_3,n_4,n_5)$ s.t. $\underline{\beta}_1$ represents only singletons and we compute $\widetilde{W}_1$ by applying $\texttt{concat}^\#(\beta_1,\widetilde{W}_0)$. In $\widetilde{W}_1$, the constraints on n_1 are the same as in $\widetilde{W}_0$, the data word variables n_3, n_4, and n_5 are removed, and the universally quantified formulas over n_2 are transformed such that n_2 represents the concatenation of the singletons denoted by n_2, n_3, n_4, and n_5 in $\widetilde{W}_0$. For example, we deduce that $\psi' := \forall y_3.\ ((y_3) \in \texttt{tl}(n_2) \wedge y_3 = 1) \Rightarrow (n_2[y_3] = 8)$.

Now we apply $\texttt{concat}^\#(n_1n_2,\widetilde{W})$. We use the fact that $\texttt{Closure}(P,\omega_1,\omega_2)[n_1/\omega_1,n2/\omega_2]$ is the union of $\mathcal{T}'_i$ with $1 \leq i \leq 5$ where

$$\mathcal{T}_1 = \{g_1 ::= (y_1) \in \texttt{tl}(n_1) \wedge y_1 = \texttt{len}(n_1) - 1,\ g_2 ::= (y_3) \in \texttt{tl}(n_2) \wedge y_3 = 1\} \\ \mathcal{T}_2 = \{g_3 ::= (y_1,y_2) \in \texttt{tl}(n_1) \wedge y_1 <_1 y_2 \wedge y_1 = \texttt{len}(n_1) - 2\}, \mathcal{T}_3 = \{g_4 ::= P[n_1/\omega]\}, \\ \mathcal{T}_4 = \{g_5 ::= (y_2,y_3) \in \texttt{tl}(n_2) \wedge y_2 <_1 y_3 \wedge y_2 = 1\}, \mathcal{T}_5 = \{g_6 ::= P[n_2/\omega]\}.$$

The procedure $\texttt{concat}^\#(\widetilde{W},n_1,n_2)$ computes the value implied by the guard $P((y_1,y_2,y_3),n_1)$ in $\widetilde{W}'$ as $U'_P = \bigsqcup^{\mathbb{Z}}_{1\leq i\leq 5} U_{\mathcal{T}'_i}$, where E_1 is the quantifier-free part of $\widetilde{W}_1$, $U_{\mathcal{T}_1} = U_{g_1} \sqcap^{\mathbb{Z}} U_{g_2} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_2]/\texttt{hd}(n_2)]$, $U_{\mathcal{T}_2} = U_{g_3} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_3]/\texttt{hd}(n_2)]$, $U_{\mathcal{T}_3} = U_{g_4}$, $U_{\mathcal{T}_4} = U_{g_5} \sqcap^{\mathbb{Z}} E_1\,[n_1[y_1]/\texttt{hd}(n_2)]$, and $U_{\mathcal{T}_5} = U_{g_6}$.

For example, from E_1, the second conjunct of ψ, and ψ', we obtain that $U_{\mathcal{T}'_1} = n_1[y_1] = 3 \wedge n_1[y_2] = 5 \wedge n_1[y_3] = 8$, which describes a sub-sequence of the Fibonacci number series. Actually, every $U_{\mathcal{T}'_i}$ describes a sub-sequence of this series which implies that the data from the tail of n_1 is also such a sub-sequence.

4.4 Soundness and Precision

An abstract value $\widetilde{W} \in A^{\mathbb{U}}$ such that $\alpha^{\mathbb{U}}(\gamma^{\mathbb{U}}(\widetilde{W})) = \widetilde{W}$ is called *a closed abstract value*. A set of guard patterns $\mathcal{P}$ is *closed* if it equals $\texttt{Closure}(\mathcal{P}',k)$, for some $\mathcal{P}'$. A guard

pattern $P(\mathbf{y_1}, \omega_1 \ldots, \mathbf{y_q}, \omega_q)$ with $\mathbf{y_i} = y_i^1 \ldots y_i^{p_i}$, for any $1 \leq i \leq q$, is called *simple* if it is of the form: $\bigwedge_{1 \leq i \leq q} \mathbf{y_i} \in \mathtt{tl}(\omega_i) \wedge y_i^1 \leq y_i^2 \leq \ldots \leq y_i^{p_i}$. Based on these definitions, the soundness and the precision of the abstract transformers are given by the following theorem. The precision of all the transformers except $\mathtt{updFst}^{\#}$ is obtained, for example, using the Octahedra [4] or the Polyhedra domain [7].

Theorem 2. *Let $\mathcal{A}_{\mathbb{U}}$ be an abstract domain as above parametrized by a set of guard patterns $\mathcal{P}$ and by a numerical abstract domain $\mathcal{A}_{\mathbb{Z}}$ which contains a sound projection operator and an exact meet operator (i.e., $\gamma^{\mathbb{Z}}(E \sqcap^{\mathbb{Z}} E') = \gamma^{\mathbb{Z}}(E) \cap \gamma^{\mathbb{Z}}(E')$). Then,*

- *All the abstract transformers of $\mathcal{A}_{\mathbb{U}}$ are sound.*
- *If (1) $\mathcal{P}$ is closed and it contains only simple patterns (2) the projection operator from $\mathcal{A}_{\mathbb{Z}}$ is exact, and (3) the abstract transformers in $\mathcal{A}_{\mathbb{Z}}$ corresponding to assignments $x = z_1 + \ldots + z_t$, $x = z_1 - 1$, and $x = 1$, where $x, z_1, \ldots, z_t$ are integer variables, are exact, then $\alpha^{\mathbb{U}}(F(Param, \gamma^{\mathbb{U}}(\widetilde{W}))) = F^{\#}(Param, \widetilde{W})$, for any F except* `updFst`*, for any set of parameters $Param$, and for any closed $\widetilde{W}$. Moreover, if the abstract transformer in $\mathcal{A}_{\mathbb{Z}}$ is exact for data expressions of the form dt, then $\mathtt{updFst}(x, dt, \widetilde{W})$ is a best abstraction for any closed $\widetilde{W}$.*

The full version [2] contains a procedure that computes for any abstract value $\widetilde{W} \in A^{\mathbb{U}}$, a closed abstract value $\widetilde{W}'$ s.t. $\gamma^{\mathbb{U}}(\widetilde{W}) = \gamma^{\mathbb{U}}(\widetilde{W}')$. Moreover, we show that for simple patterns, all the abstract transformers preserve the closure property, that is, they output closed values when applied to closed values.

5 Experimental Results

We have implemented the abstract reachability analysis using the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ domain in a tool called CINV[1]. Our implementation is generic in three dimensions. First, $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ is interfaced with the APRON platform [13], in order to use its fix-point computation engines; we use INTERPROC. Second, the implementations of the $\mathbb{DW}$-domains can be plugged in the $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{W}})$ domain. We have implemented the $\mathbb{DW}$-domain $\mathcal{A}_{\mathbb{U}}$ for the set of patterns $y \in \mathtt{tl}(w)$, $(y_1, y_2) \in \mathtt{tl}(w) \wedge y_1 < y_2$, and $y_1 \in \mathtt{tl}(w_1) \wedge y_2 \in \mathtt{tl}(w_2)$. In addition, we have implemented $\mathcal{A}_{\Sigma}$ and $\mathcal{A}_{\mathbb{M}}$ for reasoning about sums and multisets. Third, the implemented $\mathbb{DW}$-domains are generic on the numerical domain $\mathcal{A}_{\mathbb{Z}}$ used to represent data and length constraints. For this, we use again the APRON interface to access domains like octagons or polyhedra.

We have carried out experiments on a wide spectrum of programs including programs performing list traversal to search or to update data, programs with destructive updates and changes in the shape (e.g., list dispatch or reversal, sorting algorithms such as insertion sort), and programs computing complex arithmetical relations. Our tool was able to synthesize ordering constraints, data preservation constraints like those in (B) and (C) from Section 1, relations between data and lengths of lists, e.g. (D), and complex arithmetical relations, e.g. (F). Besides constraints which affect only one list, CINV was able to synthesize relations on data from different lists. For example, the program that creates a copy of a list, generates the post-condition given in (H) from Section 4.

[1] A detailed presentation is available at `http://www.liafa.jussieu.fr/cinv/`

Performances: Each example has been carried out in less than 1 second using 4KB to 63MB on an Intel 686 with 2GHz and 1 Go of RAM. The most expensive example is the insertion sort (with destructive updates) which takes 0.99s and 62.2MB. Traversal algorithms such as search and local update algorithms, require only few hundredths of a second, e.g., 0.02s for the maximum calculation. Properties of programs such as `Fibonacci` are generated in few tenths of seconds, e.g., 0.42s for (F).

6 Conclusion and Related Work

We have defined powerful invariant synthesis techniques for a significant class of programs manipulating dynamic lists with unbounded data. Future work includes (1) extending the framework to handle a wider class of data structures, e.g. doubly-linked lists, composed data structures, (2) developing heuristic techniques for automatic synthesis of the patterns used in $\mathcal{A}_{\mathbb{U}}$, and (3) defining other abstract domains for data sequences, in particular, domains based on different classes of universally quantified formulas.

Related Work: Invariant synthesis for programs with dynamic data structures has been addressed using different approaches including abstract interpretation [8–12, 16–18], Craig interpolants [14], and automata-theoretic techniques [1, 3]. The contributions of our paper are (1) a generic framework for combining an abstraction for the heap with various abstraction for data sequences, (2) new abstract domains on data sequences to reason about aspects beyond the reach of the existing methods such as the sum or the multiset of all elements in a sequence, as well as a new domain for generating an expressive class of first order universal formulas, and (3) precision results of the abstract transformers for a significant class of programs. Several works [8, 12, 16] consider invariant synthesis for programs with uni-dimensional arrays of integers. These programs can be straightforwardly encoded in our framework. In [11], a synthesis technique for universally quantified formulas is presented. Our technique differs from this one by the type of user guiding information. Indeed, the quantified formulas considered in [11] are of the form $\forall \mathbf{y}.\, F_1 \Rightarrow F_2$, where F_2 must be given by the user. In contrast, our approach fixes the formulas in left hand side of the implication and synthesizes the right hand side. Therefore, the two approaches are in principle incomparable. The techniques in [8, 12] are applicable to programs with arrays. The class of invariants they can generate is included in the one handled by our approach using $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$. These techniques are based on an automatically generated finite partitioning of the array indices. We consider a larger class of programs for which these techniques can not be applied. The analysis introduced in [16] for programs with arrays can synthesize invariants on multisets of the elements in array fragments. This technique differs from ours based on the domain $\mathcal{A}_{\mathbb{HS}}(\mathcal{A}_{\mathbb{U}})$ by the fact that it can not be applied directly to programs with dynamic lists. Finally, the analysis in [10] combines a numerical abstract domain with a shape analysis. It is not restricted by the class of data structures but it considers only properties related to the shape and to the size of the memory, assuming that data have been abstracted away. Our approach is less general concerning shape properties but it is more expressive concerning properties on data.

References

1. Bouajjani, A., Bozga, M., Habermehl, P., Iosif, R., Moro, P., Vojnar, T.: Programs with lists are counter automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 517–531. Springer, Heidelberg (2006)
2. Bouajjani, A., Dragoi, C., Enea, C., Rezine, A., Sighireanu, M.: Invariant synthesis for programs manipulating lists with unbounded data. Research report 00473754, HAL (2010)
3. Bozga, M., Habermehl, P., Iosif, R., Konecný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009)
4. Clarisó, R., Cortadella, J.: The octahedron abstract domain. In: Giacobazzi, R. (ed.) SAS 2004. LNCS, vol. 3148, pp. 312–327. Springer, Heidelberg (2004)
5. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252 (1977)
6. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proc. of POPL, pp. 269–282 (1979)
7. Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proc. of POPL, pp. 84–96 (1978)
8. Gopan, D., Reps, T.W., Sagiv, S.: A framework for numeric analysis of array operations. In: Proc. of POPL, pp. 338–350 (2005)
9. Gotsman, A., Berdine, J., Cook, B.: Interprocedural shape analysis with separated heap abstractions. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 240–260. Springer, Heidelberg (2006)
10. Gulwani, S., Lev-Ami, T., Sagiv, M.: A combination framework for tracking partition sizes. In: Proc. of POPL, pp. 239–251 (2009)
11. Gulwani, S., McCloskey, B., Tiwari, A.: Lifting abstract interpreters to quantified logical domains. In: Proc. of POPL, pp. 235–246 (2008)
12. Halbwachs, N., Péron, M.: Discovering properties about arrays in simple programs. In: Proc. of PLDI, pp. 339–348 (2008)
13. 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)
14. 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)
15. Manevich, R., Yahav, E., Ramalingam, G., Sagiv, S.: Predicate abstraction and canonical abstraction for singly-linked lists. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 181–198. Springer, Heidelberg (2005)
16. Perrelle, V., Halbwachs, N.: An analysis of permutations in arrays. In: Barthe, G. (ed.) VMCAI 2010. LNCS, vol. 5944, pp. 279–294. Springer, Heidelberg (2009)
17. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM Trans. Program. Lang. Syst. 24(3), 217–298 (2002)
18. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)

Termination Analysis with Compositional Transition Invariants*

Daniel Kroening[1], Natasha Sharygina[2,4],
Aliaksei Tsitovich[2], and Christoph M. Wintersteiger[3]

[1] Oxford University, Computing Laboratory, UK
[2] Formal Verification and Security Group, University of Lugano, Switzerland
[3] Computer Systems Institute, ETH Zurich, Switzerland
[4] School of Computer Science, Carnegie Mellon University, USA

Abstract. Modern termination provers rely on a safety checker to construct disjunctively well-founded transition invariants. This safety check is known to be the bottleneck of the procedure. We present an alternative algorithm that uses a light-weight check based on transitivity of ranking relations to prove program termination. We provide an experimental evaluation over a set of 87 Windows drivers, and demonstrate that our algorithm is often able to conclude termination by examining only a small fraction of the program. As a consequence, our algorithm is able to outperform known approaches by multiple orders of magnitude.

1 Introduction

Automated termination analysis of systems code has advanced to a level that permits industrial application of termination provers. One possible way to obtain a formal argument for termination of a program is to *rank* all states of the program with natural numbers such that for any pair of consecutive states s_i, s_{i+1} the rank is decreasing, i.e., $rank(s_{i+1}) < rank(s_i)$. In other words, a program is terminating if there exists a *ranking function* for every program execution.

Substantial progress towards the applicability of procedures that compute ranking arguments to industrial code was achieved by an algorithm called *Binary Reachability Analysis (BRA)*, proposed by Cook, Podelski, and Rybalchenko [1]. This approach combines detection of ranking functions for program paths with a procedure for checking safety properties, e.g., a Model Checker. The key idea of the algorithm is to encode an intermediate termination argument into a program annotated with an assertion, which is then passed to the safety checker. Any counterexample for the assertion produced by the safety checker contains a path that violates the intermediate termination argument. The counterexample path is then used to compute a better termination argument with the help of methods that discover ranking functions for program paths.

* Supported by the Swiss National Science Foundation under grant no. 200020-122077, by EPSRC grant no. EP/G026254/1, by the EU FP7 STREP MOGENTES, and by the Tasso Foundation.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 89–103, 2010.

A broad range of experiments with different implementations have shown that the bottleneck of this approach is the safety check [1,2]: Cook et al. [1] report more than 30 hours of runtime for some of their benchmarks. The time for computing the ranking function for a given program path is insignificant in comparison. Part of the reason for the difficulty of the safety checks is their dual role: they ensure that a disjunctively composed termination argument is correct and they need to provide sufficiently deep counterexamples for the generation of further ranking arguments.

We propose a new algorithm for termination analysis that addresses these challenges as follows: 1) We use a light-weight criterion for termination based on *compositionality* of transition invariants. 2) Instead of using full counterexample paths, the algorithm applies the path ranking procedure directly to increasingly deep unwindings of the program until a suitable ranking argument is found. We prove soundness and completeness (for finite-state programs) of our approach and support it by an extensive evaluation on a large set of Windows device drivers. Our algorithm performs up to 3 orders of magnitude faster than BRA, as it avoids the bottleneck of safety checking in the iterative construction of a termination argument.

2 Background

Preliminaries. We define notation for programs and record some basic properties we require later on. Programs are modeled as *transition systems.*

Definition 1 (Transition System). *A transition system (program) P is a three tuple $\langle S, I, R\rangle$, where*

- *S is a (possibly infinite) set of states,*
- *$I \subseteq S$ is the set of initial states,*
- *$R \subseteq S \times S$ is the transition relation.*

A *computation* of a transition system is a maximal sequence of states s_0, s_1, ... such that $s_0 \in I$ and $(s_i, s_{i+1}) \in R$ for all $i \geq 0$. A program is terminating iff all computations of the program eventually reach a final state. The non-reflexive transitive closure of R is denoted by R^+, and the reflexive transitive closure of R is denoted by R^*. The set of reachable states is $R^*(I)$.

Podelski and Rybalchenko [3] use *Transition Invariants* to prove termination of programs:

Definition 2 (Transition Invariant [3]). *A* transition invariant *T for program $P = \langle S, I, R\rangle$ is a superset of the transitive closure of R restricted to the reachable state space, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T$.*

A well-founded relation is a relation that does not contain infinite descending chains. Podelski and Rybalchenko define a weaker notion as follows:

Definition 3 (Disjunctive Well-foundedness [3]). *A relation T is* disjunctively well-founded (d.wf.) *if it is a finite union $T = T_1 \cup \ldots \cup T_n$ of well-founded (wf.) relations.*

A program is terminating if it does not have infinite computations, and Podelski and Rybalchenko show that disjunctive well-foundedness is enough to prove termination of a program:

Theorem 1 (Termination [3]). *A program P is terminating iff there exists a d.wf. transition invariant for P.*

The literature presents a broad range of methods to obtain transition invariants. Usually, this is accomplished via synthesis of *ranking functions*, which define well-founded *ranking relations* [2,4,5,6]. We refer to such methods as *ranking procedures*.

Binary Reachability Analysis [1]. Theorem 1 gives rise to an algorithm for proving termination that constructs a d.wf. transition invariant in an incremental fashion. Initially, an empty termination argument is used, i.e., $T_0 = \emptyset$. Then, a Model Checker is used to search the reachable state space for a counterexample to termination argument T_i. If there is none, termination is proven. Otherwise, let π be the counterexample path. The counterexample may be genuine, i.e., demonstrate a prefix of a non-terminating computation. Otherwise, a well-founded relation T that includes π is constructed (via a ranking procedure). Finally, the current termination argument is updated, i.e., $T_{i+1} = T_i \cup T$ and the process is repeated.

This principle has been put to the test in various tools, most notably in Terminator [1], ARMC [7], and in an experimental version of SatAbs [2].

3 Compositional Termination Analysis

The literature contains a broad range of reports of experiments with multiple implementations that indicate that the bottleneck of Binary Reachability Analysis is that the safety checks are often difficult to decide by means of the currently available software Model Checkers [1,2]. This problem unfortunately applies to both cases of finding a counterexample to an intermediate termination argument and to proving that no such counterexample exists.

As an example, consider a program that contains a trivial loop. The d.wf. transition invariant for the loop can be constructed in a negligible amount of time, but the computation of a path to the beginning of the loop may already exceed the computational resources available.

In this section, we describe a new algorithm for proving program termination that achieves the same result while avoiding excessively expensive safety checks.

We first define the usual relational composition operator $\circ$ for two relations $A, B : S \times S$ as

$$A \circ B := \{(s, s') \mid \exists s''.(s, s'') \in A \wedge (s'', s') \in B\} \ .$$

Note that a relation R is transitive if it is closed under relational composition, i.e., when $R \circ R \subseteq R$. To simplify presentation, we also define $R^1 := R$ and $R^n := R^{n-1} \circ R$ for any relation $R : S \times S$.

While d.wf. transition invariants are not in general well-founded, there is a trivial subclass for which this is the case:

Definition 4 (Compositional Transition Invariant). *A d.wf. transition invariant T is called* compositional *if it is also transitive, or equivalently, closed under composition with itself, i.e., when $T \circ T \subseteq T$.*[1]

A compositional transition invariant is of course well-founded, since it is an inductive transition invariant for itself [3]. Using this observation and Theorem 1, we conclude:

Corollary 1. *A program P terminates if there exists a compositional transition invariant for P.*

In Binary Reachability Analysis, the Model Checker needs to compute a counterexample to an intermediate termination argument, which is often difficult. The counterexample begins with a *stem*, i.e., a path to the entry point of the loop. For many programs, the existence of a d.wf. transition invariant does not actually depend on the entry state of the loop. For example, termination of a trivial loop that increments a variable i to a given upper limit u does not actually depend on the initial value of i, nor does it depend on u. The assurance of progress towards u is enough to conclude termination.

The other purpose of the Model Checker in BRA is to check that a candidate transition invariant indeed includes R^+ restricted to the reachable states. To this end, we note that the (non-reflexive) transitive closure of R is essentially an unwinding of program loops:

$$R^+ = R \cup (R \circ R) \cup (R \circ R \circ R) \cup \ldots = \bigcup_{i=1}^{\infty} R^i \ .$$

Instead of searching for a d.wf. transition invariant that is a superset of R^+, we can therefore decompose the problem into a series of smaller ones. We consider a series of loop-free programs in which R is unwound k times, i.e., the program that contains the transitions in $R^1 \cup \ldots \cup R^k$.

Observation 2. *Let $P = \langle S, I, R \rangle$ and $k \geq 1$. If there is a d.wf. T_k with $\bigcup_{j=1}^{k} R^j \subseteq T_k$ and T_k is also transitive, then T_k is a compositional transition invariant for P.*

Proof. We show that T_k is a transition invariant for P, i.e., $R^+ \cap (R^*(I) \times R^*(I)) \subseteq T_k$. Let $(x, x') \in R^+ \cap (R^*(I) \times R^*(I))$. There must exist a path over R-edges from x to x'. Let l be the length of the path, i.e., $(x, x') \in R^l$. Note that $R \subseteq T_k$, and thus, $R^l \subseteq T_k^l$. As T_k is transitive, $T_k^l \subseteq T_k$. □

This suggests a trivial algorithm that attempts to construct d.wf. relations T_i for incrementally deep unwindings of P until it finally finds a transitive T_k, which

[1] We use the term *compositional* instead of *transitive* for transition invariants in order to comply with the terminology in the existing literature [3].

proves termination of P. However, this trivial algorithm need not terminate, even for simple inputs. This is due to the fact that T_i does not necessarily have to be different from T_{i-1}, in which case the algorithm will never find a compositional transition invariant.

We provide an alternative that does not suffer from this limitation and takes advantage of the fact that most terminating loops encountered in practice have transition invariants with few disjuncts. To present this algorithm, we require the following lemma, which enables us to exclude computations from the program that we have already proven terminating in a previous iteration:

Lemma 1. *Let $P = \langle S, I, R\rangle$ and $k \geq 1$. Let $T_1, \ldots, T_k$ be a sequence of d.wf. relations such that each is a superset of the respective $\bigcup_{j=1}^{i} R^j$ restricted to reachable transitions that are not contained in any previous T_j, i.e.,*

$$\bigcup_{j=1}^{i} R^j \setminus \bigcup_{j=1}^{i-1} T_j \cap (R^*(I) \times R^*(I)) \subseteq T_i \; .$$

If $Q := \bigcup_{i=1}^{k} T_i$ is transitive, then Q is a compositional transition invariant for the program P.

Proof. We have $\bigcup_{i=1}^{k} R^i \subseteq \bigcup_{i=1}^{k} T_i = Q$ and in particular $R \subseteq Q$. Therefore $R^+ \subseteq Q^+$ and since Q is transitive it follows that $R^+ \subseteq Q$. It is d.wf. as it is a finite union of d.wf. relations. □

As an optimization, we may safely omit some of the T_i while searching for a transitive T_k:

Lemma 2 (Optimization). *Let $T_0, \ldots, T_k$ be the sequence of d.wf. relations for application of Lemma 1. The claim of the lemma holds even if some of the $T_1, \ldots, T_{k-1}$ are not provided (empty).*

Proof. We show that Q is a transition invariant for P. Let $(x, x') \in R^+ \cap (R^*(I) \times R^*(I))$. As in the proof of Obs. 2, $(x, x') \in R^l$ for some l. The claim holds trivially for $l \leq k$ as $\bigcup_{i=1}^{k} R^i \subseteq Q$. For $l > k$, note that $(x, x') \in (R^{jk} \circ R^{l-jk})$ and $0 \leq l - jk < k$ for some $j \geq 1$. Note that $R^{jk} \subseteq Q^j$ and $R^{l-jk} \subseteq Q$. Thus, $(x, x') \in (Q^j \circ Q) = Q^{j+1}$. As Q is transitive, $Q^{j+1} \subseteq Q$, and thus $(x, x') \in Q$.

The proof of Lemma 1 still applies. As an example, our implemenation only uses those T_i where i is a power of two.

The procedure that we draw from Lemma 2 is Algorithm 1, and we call it *Compositional Termination Analysis* (CTA). This algorithm makes use of an external ranking procedure called *rank*, which generates a d.wf. ranking relation for a given set of transitions, or alternatively a set $C \in S$ of states such that $R^*(C)$ contains infinite computations. We say that *rank* is sound if it always returns either a d.wf. superset of its input or a non-empty set of states C, and we call it complete if it terminates on every input.

```
   input  : P = ⟨S, I, R⟩
   output : 'Terminating' / 'Non-Terminating'
 1 begin
 2     T := ∅;
 3     X := S;
 4     i := 1;
 5     while true do
 6         ⟨T_i, C⟩ := rank ((⋃_{j=1}^{i} R^j \ T) ∩ (X × X));
 7         if C ∩ R*(I) ≠ ∅ then
 8             return 'Non-Terminating';
 9         else if C = ∅ and T ∪ T_i is transitive then
10             return 'Terminating';
11         else
12             X := X \ C;
13             T := T ∪ T_i;
14             i := i + j, where j > 0;
15         end
16     end
17 end
```

Algorithm 1. Compositional Termination Analysis

Algorithm 1 maintains a set $X \subseteq S$ that is an over-approximation of the set of reachable states, i.e., $R^*(I) \subseteq X$. It starts with $X = S$ and at $i = 1$. It iterates over i and generates d.wf. ranking relations T_i for the transitions in $\bigcup_{j=1}^{i} R^j \setminus T$. As long as such relations are found, they are added to T. Once it finds a transitive T, the algorithm stops, as P terminates according to Lemma 2. When ranking fails for some i, the algorithm checks whether there is a reachable state in C, in which case $R^*(C)$ contains a counterexample to termination and the algorithm consequently reports P as non-terminating. Otherwise, it removes C from X, which represents a refinement of the current over-approximation of the set of reachable states.

Theorem 3. *Assuming the sub-procedure rank is sound, Algorithm 1 is sound.*

Proof. When the algorithm terminates with 'terminating' (line 10), the sequence of relations T_i constructed so far is suitable for application of Lemma 2, which proves termination. It is easy to see that the set $R^*(I)$ in Lemma 2 can be over-approximated to X. If the algorithm returns 'non-terminating' at line 8, it has found a set of reachable states from which infinite computations exist, i.e., there is a concrete counterexample to termination. □

Lines 12–14 ensure progress between iterations by excluding unreachable states (C) from the approximation X and adding the most recently found T_i in T. However, for non-terminating input programs, the algorithm may not terminate for two reasons: a) *rank* is not required to terminate, and b) there may be an infinite sequence of iterations. This is not the case for finite S if the input program is non-terminating, since sound and complete ranking procedures exist (e.g., [5,2]) and progress towards the goal can thus be ensured:

Corollary 2. *If the sub-procedure rank is sound and complete for finite-state programs, then Algorithm 1 is sound and complete for non-terminating finite-state programs.*

Proof. We assume a non-terminating input program $P = \langle S, I, R \rangle$. As S is finite there must exist a looping counterexample with a finite stem. In each iteration, either T increases or X decreases, as $C \cap X = \emptyset$. Thus, the algorithm will eventually consider an unwinding long enough to contain the stem, at which point *rank* returns a C with $C \cap R^*(I) \neq \emptyset$ (since it is sound and complete). In both cases, progress is ensured because *rank* always returns a d.wf. ranking relation or a non-empty set C. In the worst case, the number of iterations is equal to the length of the shortest counterexample to termination. □

Note that the algorithm is not complete for terminating programs even if they are finite-state. This is due to the fact that T is not guranteed to ever become transitive, even if it contains R^+.

Example. We demonstrate the advantage of our approach over BRA on the following simple program, where $*$ represents non-deterministic choice.

```
integer i;

while i<255 do begin
  if * then i:=i+1;
  else i:=i+2;
end
```

The state space in this example is $S = \mathbb{N}_0$, and i is the only variable. A suitable wf. transition invariant is $\{(i, i') \in S^2 \mid i < i' \wedge i' \leq 256\}$, which is easily generated within a negligible amount of time. BRA subsequently needs to verify the absence of further counterexamples, which requires 14 refinement iterations when the SATABS engine is used. Compositional Termination Analysis returns immediately after synthesizing the ranking function, because the corresponding relation is transitive. A different ranking procedure may return a d.wf. transition invariant with one disjunct for each path through the loop body. In this case, our algorithm stops in the second iteration, because there are no more transitions that are not included in either of the two disjuncts.

Remark. The check in line 9 of the algorithm corresponds to checking whether $T \circ T \subseteq T$ for some relation $T : S \times S$. This corresponds to checking validity of

$$\forall x, y \in S.\ (x, y) \in T \rightarrow (x, y) \in T \circ T\ ,$$

which, in the case of symbolically-represented relations, can be established using one call to a suitable decision procedure.

4 Implementation

We have implemented Compositional Termination Analysis for ANSI-C programs. Our implementation follows Algorithm 1. It instruments the program

with termination assertions as described by Cook et al. [1] and subsequently applies the termination analysis once to each loop in the program. There are two additional features that need discussion, namely our abstracting loop slicer, and the blockwise ranking procedure.

4.1 Slicing and Loop Abstraction

To reduce the resource requirements of the Model Checker, our implementation analyzes each loop separately. It generates an inter-procedural slice [8] of the program, slicing backwards from the termination assertion. In addition, we rewrite the program into a single-loop program, abstracting from the behavior of all other loops.

Following the hypothesis that loop termination rarely depends on complex variables that are possibly calculated by other loops, our slicing algorithm replaces all assignments that depend on five or more variables with non-deterministic values. Also all loops other than the analyzed one are 'havocked': they are replaced by program fragments that assign non-deterministic values to all variables that might change during the execution of the loop (similar to the loop summarization in [9]).

Note that this is a purely practical issue: The benchmarks we use require far too much time to run without this abstraction. We have noticed however, that the abstraction is almost always precise enough, i.e., we loose only very few termination proofs. Of course, we use the exactly same slices for all methods that we compare in our evaluation.

4.2 Blockwise Ranking

The sub-procedure *rank* in Algorithm 1 may be implemented in various ways. For example, it is possible to enumerate all paths through $\bigcup_{j=0}^{i} R^j$ and to obtain a d.wf. ranking relation for every path separately. To avoid this enumeration, we employ the symbolic execution engine of CBMC [10] to find paths through the program that are not yet included in the candidate transition invariant. For this purpose, we create a temporary program that first initializes all variables with non-deterministic values, saves the state, and then executes R^i, which is loop-free. Finally, we check for inclusion of the loop pre- and post-states in the current candidate transition invariant (starting with the empty set).

If a counterexample is found, we extract a path from it and try to compute a wf. ranking relation for it. If this succeeds, this relation is added disjunctively to the current (d.wf.) candidate transition invariant. This procedure is equivalent to the application of Binary Reachability Analysis to a loop-free program fragment.

The explicit check for compositionality of the candidate transition invariant can often be avoided. If we find that T is composed of a single wf. ranking relation, we trivially know that the transition relation is also well-founded, since it is a subset of a wf. transition invariant.

```
1  void main()
2  {
3   int x;
4   int debug = 0;
5
6   while(x<255) {
7    if (x%2!=0) x--;
8    else x+=2;
9    if (debug!=0) x=0;
10  }
11 }
```

Fig. 1. A loop with four paths through its body

The synthesis of ranking relations for paths is out of the scope of this paper, but we would like to point the interested reader to some recent results in this area [2,4,5].

4.3 Illustration

To illustrate Compositional Termination Analysis and our implementation, we demonstrate the most important steps on a simple program. The source code for this demonstration is given in Figure 1 and it consists of an ANSI-C program that contains a single loop with four paths through its body. Figure 2 shows the control flow graph of this program and defines the program locations l_1 to l_9.

Our algorithm starts at $i = 1$ and R^1 is equivalent to a single unwinding of the loop, i.e., a single copy of the loop body. The initial value of X is S, which allows any entry state of the loop, including states that have the variable *debug* set to values other than 0. The initial termination argument T is $\emptyset$. The procedure *rank* analyzes R^1 and, since T is empty, any path between the locations l_2 and l_8 violates the current termination argument. Consider the path passing through locations $l_2, l_3, l_4, l_6, l_7, l_8$. There is no wf. ranking relation for this path because the segment between locations l_7 and l_8 sets x to 0. This means that x is always set to the same value, which also happens to satisfy the loop entry condition. Furthermore, the variable *debug* never changes its value. Thus, the procedure *rank* returns a non-empty path precondition $C \equiv (debug \neq 0) \wedge (x < 255) \wedge (x\%2 \neq 0)$. However, C does not contain any reachable loop entry state in the original program because *debug* is set to 0 between l_1 and l_2. Consequently, the test at line 7 of Algorithm 1 fails and X is updated to $X \setminus C$ at line 12 of Algorithm 1.

The algorithm continues with a refined X, while T is still empty. There exist two more paths through the block R^1: l_2, l_3, l_4, l_6, l_8 and l_2, l_3, l_5, l_6, l_8. The procedure *rank* finds a ranking function for each of them, namely $+x$ for the first path and $-x$ for the second, and constructs the d.wf. ranking relation $T_1 \equiv x < x' \vee -x < -x'$, which is disjuncively composed of two ranking relations over the pre- and post-state of the loop (x and x', respectively).

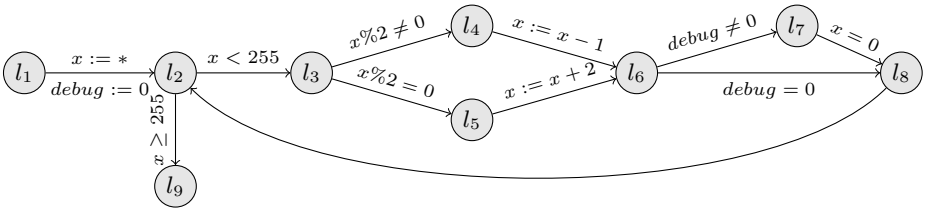


Fig. 2. Control-flow graph of the program in Fig. 1

The constructed d.wf. ranking relation T_1 is added to the termination argument T and i is increased. Since the d.wf. ranking relation found in the previous iteration was disjunctive, the algorithm proceeds to the next iteration, where *rank* examines R^2, i.e., it now explores two loop unwindings. However, it cannot find any new path that is both in R^2 and not included in T. Therefore, the algorithm concludes that the program in Fig. 1 terminates according to Lemma 2.

5 Experimental Results

We have evaluated our implementation of Compositional Termination Analysis on a set of 87 Windows device drivers taken from the Windows Device Driver Kit.[2] Every driver is analyzed in two different configurations, which results in a total of 174 benchmarks. We use GOTO-CC[3] to extract control flow graphs from the original sources, which are then passed to our Compositional Termination Analysis engine.

We compare our implementation to an implementation of Binary Reachability Analysis using SATABS as the safety checker. In all our experiments, we use a simple and incomplete coefficient enumeration approach to synthesize polynomial ranking functions using a SAT solver. This and our implementation of Binary Reachability Analysis have been used in a recent comparison of ranking engines [2]. For our evaluation we run Binary Reachability Analysis on every driver using a timeout of 2 hours and a memory limit of 2 GB on an Intel Xeon 3 GHz machine. All those loops that this engine analyzes successfully within the time limit serve as the baseline for our comparison. Note that some loops may not require calls to a ranking engine, either because they are unreachable or termination is trivial and shown by preprocessing. We have excluded those loops, i.e., our evaluation is only on loops that require a ranking engine at least once. Our baseline consists of 99 terminating and 45 non-terminating benchmarks.

Whenever the ranking engine is not able to find a valid d.wf. transition invariant for a block, it returns the weakest precondition of a corresponding path. This precondition describes a set of states from which termination of the program is not guaranteed. Our implementation can be configured to react to this situation

[2] Version 6, available online at `http://www.microsoft.com/whdc/devtools/wdk/`
[3] `http://www.cprover.org/goto-cc/`

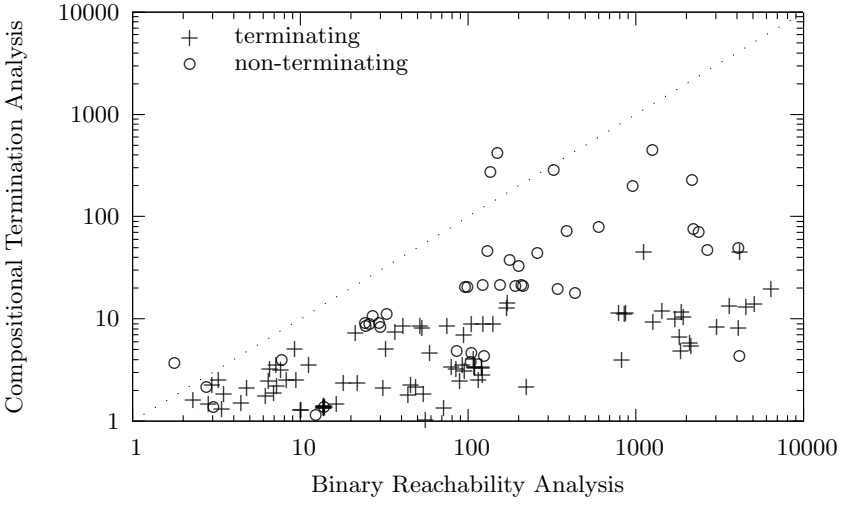


Fig. 3. Experimental results using path precondition checks

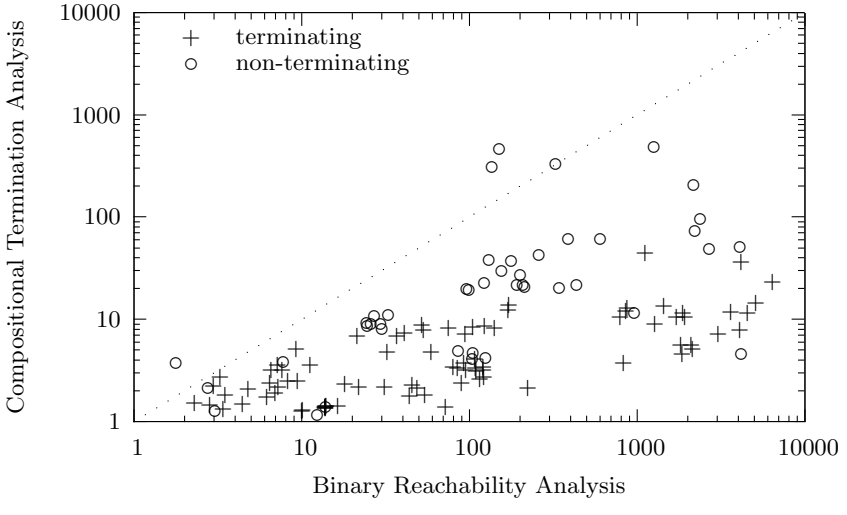


Fig. 4. Experimental results when using loop reachability checks

in three different ways: a) check reachability of the precondition, b) check reachability of the loop, or c) report the loop as non-terminating. We present results for all three variants.

First, we discuss the results obtained from variant a), which checks path preconditions using a Model Checker (SATABS in our implementation) and thus features the same level of precision as Binary Reachability Analysis. Every data point in Figure 3 represents one loop. On the horizontal axis we indicate the total time taken to analyze this loop using Binary Reachability Analysis. The

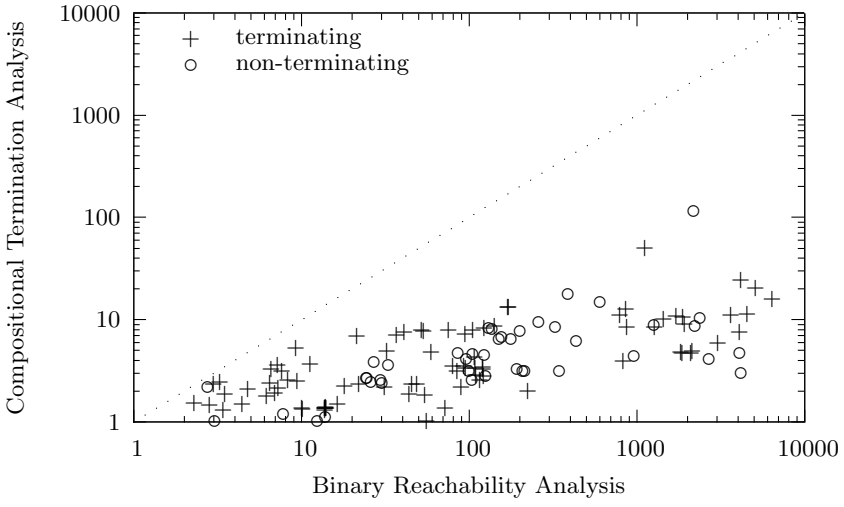


Fig. 5. Experimental results without loop reachability or precondition checks

vertical axis indicates the time taken by Compositional Termination Analysis. As apparent in Figure 3, Compositional Termination Analysis is up to three orders of magnitude faster. The average speedup factor is 52. However, there are a few non-terminating benchmarks on which it is slower. This is due to the fact that on non-terminating loops many (or all) path preconditions are eventually enumerated. The resulting loop-free programs are sometimes too difficult for the model checker. A possible solution for this problem are techniques that compute a more general precondition of non-termination. A recent technique described by Cook et al. [11], which constructs preconditions of termination, could be applied for this purpose.

Figure 4 provides the results obtained when checking for general loop reachability, which is essentially a crude over-approximation of the precondition of the non-terminating paths through the loop. The results are very similar to those of the previous variant. This is due to the fact that most loops are indeed reachable and so are most path preconditions. There is no difference in precision compared to variant a) on these benchmarks, i.e., no termination proofs are lost compared to an actual precondition check.

Finally, we discuss the results obtained with variant c), which reports non-termination immediately, i.e., without checking reachability of the loop or a precondition. Naturally, this version of our algorithm is the fastest (Fig. 5). The imprecision introduced by not checking loop reachability or path preconditions does not have any effect on these benchmarks.

Figure 6 shows that the overall capacity of Compositional Termination Analysis is much higher than that of Binary Reachability Analysis: At an equal level of precision, Compositional Termination Analysis is able to analyze more than three times the number of benchmarks that Binary Reachability Analysis is able to analyze.

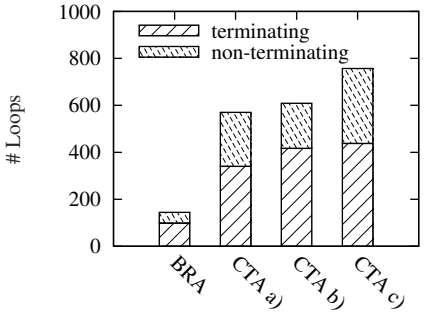


Fig. 6. Total number of loops analyzed within 2 hours per driver

All our experimental data, the implementation, and additional material is available for further research at `http://www.cprover.org/termination/`.

6 Related Work

Termination analysis has its roots in the work of Turing [12,13]. Since then, substantial progress has been made in various areas of computer science: logic programming (e.g., [14]), term rewriting-based analysis (e.g., [15]), and functional programming (e.g., [16]).

We make use of a sub-procedure for ranking individual paths [4,17,6]; this problem is orthogonal to our contribution, which is focused on the iterative construction of a termination argument for a full program. We elaborate on the differences of our new algorithm and BRA as described in [3,18,1,11].

The basis for reasoning about transition invariants, including the result that d.wf. transition invariants can be used to show termination, has been presented in [3]. The BRA algorithm was presented in [1]. We also make use of the results of [3], but develop them in a different direction: we show how to prove termination using the compositionality of transition invariants. Our algorithm passes smaller, loop-free fragments of the program to the safety checker, which enables it to outperform Binary Reachability Analysis.

In [11], the authors under-approximate the weakest precondition of paths to find a condition for termination. This result can be exploited in the context of our algorithm as well, as it allows for a generalization of path preconditions.

Berdine et al. present an algorithm for proving termination that is based on abstract interpretation [19]. Using an invariance analysis they construct a variance analysis, and they use the fact that the transitive closure of a well-founded relation is also well-founded to show that the fixed-point obtained by their analysis is correct. Their result may be used to improve the overall performance of our algorithm, as it can be modified to generate d.wf. transition invariants via abstraction.

Biere, Artho, and Schuppan propose an encoding of liveness properties into an assertion [20]. This approach allows proving termination of programs without

a ranking sub-procedure. It has been reported to prove termination of programs that require non-linear ranking functions. Prior experimental results on our benchmarks indicate this encoding results in difficult safety checks [2].

7 Conclusion

The safety check is known as the bottleneck of Binary Reachability Analysis (BRA). We present a new algorithm for proving program termination that avoids this expensive safety check and is therefore able to outperform BRA. The latter relies on a safety checker to detect correctness of a disjunctively well-founded termination argument. We propose to check for compositionality of a candidate termination argument, which is much less expensive. To perform this test, our algorithm passes only loop-free segments of the program to a symbolic execution engine and, consequently, achieves much higher performance than other termination provers. In case the termination argument has to be refined, BRA uses a full counterexample path computed by a safety checker. In contrast, we pass an incrementally deeper unwinding of the loop to the rank finding procedure. Experimental results indicate an average speedup factor of 52 in comparison to BRA.

References

1. Cook, B., Podelski, A., Rybalchenko, A.: Termination proofs for systems code. In: PLDI, pp. 415–426. ACM, New York (2006)
2. Cook, B., Kroening, D., Ruemmer, P., Wintersteiger, C.: Ranking function synthesis for bit-vector relations. In: TACAS, pp. 236–250. Springer, Heidelberg (2010)
3. Podelski, A., Rybalchenko, A.: Transition invariants. In: LICS, pp. 32–41. IEEE Computer Society, Los Alamitos (2004)
4. 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)
5. 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. 465–486. Springer, Heidelberg (2004)
6. 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)
7. 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 (2006)
8. Horwitz, S., Reps, T.W., Binkley, D.: Interprocedural slicing using dependence graphs. In: PLDI, pp. 35–46. ACM, New York (1988)
9. Kroening, D., Sharygina, N., Tonetta, S., Tsitovich, A., Wintersteiger, C.M.: Loop summarization using abstract transformers. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 111–125. Springer, Heidelberg (2008)

10. 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)
11. 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)
12. Turing, A.M.: On computable numbers, with an application to the Entscheidungsproblem. Proc. London Math. Soc. 2, 230–265 (1936)
13. Turing, A.: Checking a large routine. In: Report of a Conference on High Speed Automatic Calculating Machines, Univ. Math. Lab., Cambridge, pp. 67–69 (1949)
14. Codish, M., Taboch, C.: A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In: Hanus, M., Heering, J., Meinke, K. (eds.) ALP 1997 and HOA 1997. LNCS, vol. 1298, pp. 31–45. Springer, Heidelberg (1997)
15. Thiemann, R., Giesl, J.: The size-change principle and dependency pairs for termination of term rewriting. Appl. Alg. in Eng., Comm. & Comp. 16, 229–270 (2005)
16. Lee, C.S., Jones, N.D., Ben-Amram, A.M.: The size-change principle for program termination. In: POPL, pp. 81–92. ACM, New York (2001)
17. Colón, M., Sipma, H.: Practical methods for proving program termination. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 442–454. Springer, Heidelberg (2002)
18. 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)
19. Berdine, J., Chawdhary, A., Cook, B., Distefano, D., O'Hearn, P.: Variance analyses from invariance analyses. SIGPLAN Not. 42, 211–224 (2007)
20. Biere, A., Artho, C., Schuppan, V.: Liveness checking as safety checking. Electr. Notes Theor. Comput. Sci. 66 (2002)

Lazy Annotation for Program Testing and Verification

Kenneth L. McMillan

Cadence Berkeley Labs

Abstract. We describe an interpolant-based approach to test generation and model checking for sequential programs. The method generates Floyd/Hoare style annotations of the program on demand, as a result of failure to achieve goals, in a manner analogous to conflict clause learning in a DPLL style SAT solver.

1 Introduction

The DPLL approach to Boolean satisfiability combines search and deduction in a mutually reinforcing way. It focuses deduction where the search becomes blocked, deducing facts that guide the search away from the failure. Here, we consider an approach to program testing and verification inspired by DPLL. As in DART [3], we use symbolic execution to search the space of execution paths. When the search fails to reach a goal, we deduce a program annotation that will prevent us from being blocked in the same way in the future. Since annotations are deduced only in response to search failures, we will call this method *lazy annotation*.

The algorithm proceeds roughly as follows. We designate a set of program locations as goals to be reached. In the following examples, the goals are calls to a function `error`. The vertices (locations) and edges (statements) of the program's control flow graph will be labeled with formulas. A label represents a condition under which no goal can be reached. Initially, there are no labels (no annotation being equivalent to false). We execute the program *symbolically* along some chosen path. Each input to the program is represented by a symbolic value p_i. In the symbolic state, each program variable is evaluated as a symbolic expression over these parameters. A *constraint* is also maintained, which accumulates the conjunction of the branch guards along the chosen path, as a function of the parameters.

We say that our state is *blocked* if the current vertex label is implied, meaning we cannot reach a goal from this state. When we are blocked, we will backtrack along the edge we just executed, annotating it with a new label that blocks that edge. When choosing a branch to execute, we are guided by these edge labels. A blocked edge cannot lead to a goal, so we always continue along an unblocked edge if there is one. When all outgoing edges are blocked in the current state, we label the current location with the conjunction of the conditions that block the outgoing edges, thus blocking the current state and causing us to backtrack.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 104–118, 2010.

As an example, consider the fragment `simple` in Figure 1. Suppose on entering this code at l_1, our symbolic state is $x = p_0$ with constraint T (true). Branching from l_1 to l_2, we obtain the constraint $p_0 = 0$. At l_2 we have the choice to branch to l_3 or l_6. Since neither edge is labeled, we choose arbitrarily l_6. Now there is only one choice and we branch to l_7, obtaining the unsatisfiable constraint $p_0 = 0 \wedge p_0 < 0$. At this point we are blocked, since F (false) holds in the current state and absence of annotation is equivalent to F.

We therefore backtrack, annotating edge $l_6 \rightarrow l_7$. In general, if the current annotation is ϕ, we can annotate the incoming edge with the weakest precondition of ϕ along that edge. In this case, the current annotation is $\phi = F$ and the weakest precondition is $x \geq 0$. After backtracking, we return to l_6. There is only outgoing edge, labeled $x \geq 0$. Therefore, we can label $l_6 : x \geq 0$. We are now blocked (since $x = 0$ in our current state) so we backtrack to l_2, labeling edge $l_2 \rightarrow l_6 : x \geq 0$ (again using the wp). Notice that each time we label an edge, that edge becomes blocked in the current symbolic state (and possibly other states).

Since the edge $l_2 \rightarrow l_3$, is still unblocked, we follow it. Our annotation has forced the search in a different direction. Moving on to l_4, we have $y = p_1$ (a new input), then at l_5 we have the constraint $p_1 \geq 0$ (corresponding to the guard $y \geq 0$). Finally we arrive at l_6 in the state $x = p_0 + p_1$ with constraints $p_0 = 0$ and $p_1 \geq 0$. Since this implies the previous annotation $l_6 : x \geq 0$, we are blocked. The fact we previously learned tells us there is no path to the goal from our current state. Backtracking to l_5 and taking the weakest precondition of $x \geq 0$ then gives us $l_5 : x + y \geq 0$. When we backtrack to l_4, however, we observe a slight problem. The weakest (liberal) precondition of $x + y \geq 0$ with respect to the assertion $y \geq 0$ is $y < 0 \vee x + y \geq 0$. However, the variable y is irrelevant here, and we could just as well block the state with $x \geq 0$ (also a precondition, not the weakest). This latter fact can be computed as an *interpolant* as we will show later. The advantage of the interpolant becomes clear in the next step when the weakest precondition would yield $l_3 : \forall y.\ (y < 0 \vee x + y \geq 0)$. We can simplify this to $x \geq 0$, but this requires quantifier elimination, which can be very expensive. By computing preconditions with interpolants, we avoid the need for quantifier elimination.

Now we backtrack to l_2, labeling the edge $(l_2 \rightarrow l_3)$ with $x \geq 0$. At this point, both edges from l_2 are blocked, so we label it with the conjunction of the blocking labels along these edges, yielding $l_2 : x \geq 0$. Finally, we label $l_1 : T$, proving that the goal `error` cannot be reached from l_1.

Lazy annotation has the advantage that, like DPLL, it can recover from irrelevant or too-specific deductions. Consider, for example, fragment `diamond`, and suppose we first execute the path through l_3 and l_7. After backtracking, we label $l_6 \rightarrow l_7 : p$. Then executing edge $l_6 \rightarrow l_9$ we are blocked, because $a = 1$ in the current state. Thus, rather than exploring this path and discovering it is also safe because p holds, we label $l_6 \rightarrow l_9$ with the irrelevant condition a, and thus we label $l_6 : p \wedge a$. However, when we return to l_6 via l_5, note what happens. The edge $l_6 \rightarrow l_7$ is blocked by label p so we follow $l_6 \rightarrow l_9$, ultimately labeling it p

```
simple:
l1  assume(x == 0);
l2  if(*){
l3      int y = *;
l4      assume(y >= 0);
l5      x = x + y;
    }
l6  if(x < 0)
l7      error();
```

```
diamond:
l1  assert(p);
l2  if(*)
l3      a = 1;
    else
l5      a = 0;
l6  if(a)
l7      x = x + 1;
l8  else
l9      x = x - 1;
l10 if(!p)
l11     error();
```

```
loop1:
l1  assert(x == y);
l2  i = 0
l3  while(i < n){
l4      x = x + 1;
l5      i = i + 1;
l6  }
l10 if(x < y)
l11     error();
```

```
    int x;

    foo(){
l1      x = x + 1;
l2      return;
    }

call1:
l3  x = y;
l4  foo();
l5  if(x < y)
l6      error();
```

```
    int x;

    rec1(){
l1      if(*){
l3          rec1();
l4          rec1();
        }
l5      x = x + 1;
l6      return;
    }
```

Fig. 1. Example program fragments

as well. Thus, we label $l_6 : p$. Location l_6 is now labeled by the *disjunction* of p and $p \wedge a$. The stronger condition $p \wedge a$ is effectively subsumed, and we discard it.

Note how this differs from partition refinement approaches such as Synergy [5] and its descendants [4]. In Synergy, after executing the path through l_7, location l_9 is on the frontier. This causes l_6 to be partitioned by the irrelevant predicate a. There is no way to recover from this irrelevant refinement. With sequences of such diamonds, we can construct reasonable scenarios in which this leads Synergy to an exponential explosion of partitions, while lazy annotation is polynomial.

Unbounded loops. Consider the simple unbounded loop in fragment `loop1`. To force this loop to terminate, we will use a simple trick: we instrument the program with a variable τ that decreases in every iteration, and we annotate all locations with $\tau < 0$, blocking any path in which τ becomes negative. We pick an arbitrary initial value of τ, say, zero. This forces the loop to be executed at most once. Say we execute first the path that skips the loop. Using interpolants, we label $l_3 \rightarrow l_{10} : x \geq y$ and backtrack into the loop. When the loop completes, we decrement τ and are thus blocked by $l_3 : \tau < 0$. We thus backtrack through the loop. Using interpolants, we obtain $l_3 \rightarrow l_4 : \tau < 1$. Thus, we label $l_3 : x \geq y \wedge \tau < 1$. Backtracking further, we label $l_1 : \tau < 1$ and terminate. We have not

proved unreachability of `error`, since our annotation depends on the bogus label $\tau < 0$. However, our annotation can be strengthened by induction. We simply plug $\tau = 0$ into our labels and see if they are inductive. We obtain $l_3 : x \geq y$ which is in fact inductive, so we keep it. We drop any non-inductive labels iteratively, obtaining the greatest inductive subset as a fixed point. Effectively we have used bounded model checking as a heuristic for constructing an inductive invariant. Note that using weakest precondition, we would have obtained $l_3 : x \geq 0 \vee i < n$ which is not inductive. To handle unbounded loops, we need some form of generalization, here provided by interpolation. If strengthening by induction fails to prove unreachability, we can increase the initial τ value and try again.

Procedure summaries. To handle programs with procedures modularly, we label them with negative summaries. This is a formula that uses primed variables to represent the exit state of the procedure, and is true when that exit state is *not* reachable. For example, if on exit the value of x must be greater than its current value, the negative summary would be $x' \leq x$. We can use lazy annotation to compute a summary for a procedure from a given initial state, with a desired post-condition ψ. This reusable summary can replace a call to the procedure in various contexts. Consider fragment `call1` in the figure. Here, we start at l_3 in state $x = p_0, y = p_1$. When we reach the call to `foo` at l_4, we recursively call lazy annotation to compute a summary of `foo` that proves the current post-condition of the call, which is $l_5 : F$. Obviously, this cannot be proved, and we obtain a counterexample, which is a path to the return state $x = p_1 + 1, y = p_1$. Continuing from l_5, we eventually label $l_5 : y \leq x$. Now when we backtrack to l_4, we recursively try to compute a summary of `foo` that proves the post-condition $y \leq x$. This time we succeed, computing the negative summary $l_1 : x' < x$ as an interpolant (see Section 2.3). Using this summary for $l_4 \rightarrow l_5$, we can label $l_4 : x \geq y$ and terminate. Moreover this same summary at l_1 may be useful in other contexts, allowing us to return immediately from `foo`.

The advantage of using interpolants to compute summaries is that we can obtain more general summaries. A method such as Smash [4] that uses weakest precondition to compute a summary with the post-condition $x \geq y$ would yield a summary such as $x \geq y \overset{\text{foo}}{\Rightarrow} x \geq y$ containing the irrelevant variable y. To be able to reuse this summary in another context, we need to be able to universally quantify over y, which again involves us in quantifier elimination. Using interpolants, this complication is avoided.

Recursion. Finally, consider the recursive function `rec1` in the figure and suppose we want to compute a summary for initial state $x = 0$ and post-condition $x \geq 0$. To force termination, we decrement τ on recursive calls, and initialize τ to 1. Now suppose we first take the path through l_3. Because τ is decremented, the recursive calls at l_3 and l_4 must take the non-recursive path, yielding an exit state $x = 2$. This satisfies the post-condition $x \geq 0$, giving a negative summary $l_5 : x' < x$. Backtracking to l_4, we again call recursively with a post-condition equivalent to $x \geq 0$ (for details, see Section 2.3) which yields a summary $l_1 : x' < x \wedge \tau < 1$. The same summary is reused in backtracking to

l_3, which eventually gives us $l_1 : x' < x \wedge \tau < 2$. Setting $\tau = 0$, we find that $l_1 : x' < x$ is inductive and terminate.

Related work. Lazy annotation is similar to lazy abstraction with interpolants [8] in that it computes an inductive invariant using only interpolation. Because it explores only feasible paths, however, it is useful for testing, and can efficiently handle bounded loops. Moreover, it handles procedures modularly. It is also similar in some respects to Synergy [5] and related methods [4] that use partition refinement. However, unlike these methods, it can recover from too-specific refinements. In fact, we can think of the annotation as partitioning each location into exactly two state sets. Lazy annotation can also compute more general summaries using interpolants.

Relative to predicate abstraction approaches [1,9], lazy annotation has the advantage that it avoids the expensive predicate image computation. However it is in another sense orthogonal to these methods, as predicate abstraction can be used to inductively strengthen the annotations obtained by lazy annotation, possibly speeding convergence for unbounded loops, while avoiding the many iterations produced by counter-example guided abstraction refinement. In fact, any backward abstract interpretation can be used for this purpose.

2 Lazy Annotation

Throughout this paper, we will use standard first-order logic (FOL) and the notation $\mathcal{L}(\Sigma)$ to denote the set of well-formed formulas (*wff's*) of FOL over a vocabulary Σ of uninterpreted symbols (the formulas may also include various interpreted symbols, such as $=$ and $+$). For a given formula ϕ, $\mathcal{L}(\phi)$ will denote the *wff's* over the uninterpreted vocabulary of ϕ. We will write $\phi[\sigma]$ to indicate that structure σ is a model of formula ϕ. To every uninterpreted symbol s, we associate a unique symbol s' (that is, s with one prime added). For any formula or term ϕ or vocabulary S, we will write ϕ' or S' for the result of adding one prime to all the non-logical symbols in ϕ or S.

Given a pair of FOL formulas (A, B), such that $A \wedge B$ is inconsistent, an *interpolant* for (A, B) is a formula $\hat{A}$ such that A implies $\hat{A}$, $\hat{A}$ implies $\neg B$, and $\hat{A} \in \mathcal{L}(A) \cap \mathcal{L}(B)$. The Craig interpolation lemma [2] states that interpolants always exist for inconsistent formulas in FOL. A variety of techniques exist for deriving an interpolant for (A, B) from refutation of $A \wedge B$ in a suitable proof system [7]. This allows us to generate interpolants using a theorem prover or proof-generating decision procedure.

Modeling programs. We assume a vocabulary S of variables representing the program's data state, a domain D of data values, and a collection of program actions $\mathcal{A}$ provided by the programming language. A *program* is a finite, rooted, labeled graph (Λ, l_0, Δ) where Λ is a finite set of program locations, $l_0 \in \Lambda$ is a distinguished initial location and $\Delta \subseteq \Lambda \times \mathcal{A} \times \Lambda$ is a set of transitions labeled by actions. Let $\mathrm{Out}(l)$ denote the set of outgoing edges from location l.

A *program path* of length k is an alternating sequence of the form $l_0 a_0 l_1 a_1 \ldots l_k$, where each triple (l_i, a_i, l_{i+1}) is in Δ. A *data state* in $\mathcal{D}$ is a map $S \to D$. We fix an initial data state d_0. The semantics of an action $a \in \mathcal{A}$, denoted $\mathrm{Sem}(a)$, is subset of $\mathcal{D} \times \mathcal{D}$. A *program run* of length k is a pair (π, σ), where π is a program path, and $\sigma = d_0 \cdots d_k$ is a sequence of data states such that for all $0 \leq i < k$, $(d_i, d_{i+1}) \in \mathrm{Sem}(a_i)$. A *state* is a pair $(l, d) \in \Lambda \times \mathcal{D}$. The reachable states are all the pairs (l_k, d_k) for some run of length k.

A *state formula* is a formula in $\mathcal{L}(S)$. A *transition formula* is a formula in $\mathcal{L}(S \cup S')$. For action a and formulas ϕ, ψ (that may contain non-program variables) the *Hoare triple* $\{\phi\}a\{\psi\}$ is valid when for every data state d_1, and interpretation $\mathcal{I}$ of the non-program variables, such that $\phi[d_1 \cup \mathcal{I}]$ and every d_2 such that $(d_1, d_2) \in \mathrm{Sem}(a)$, we have $\psi[d_2 \cup \mathcal{I}]$. We assume that $\mathrm{Sem}(a)$ can be expressed as a transition formula, which by abuse of notation, we will write $\mathrm{Sem}(a)$. Since actions and transition formulas are interchangeable, we will also write $\{\phi\}t\{\psi\}$ where t is an arbitrary transition formula.

It is useful to define a *relational join* operator for relations expressed as formulas. Let ϕ and ψ be formulas, and f be an indexed set of variables with a unique variable f_v associated to each $v \in S$. Then $\phi \times_f \psi$ is the formula $\phi\langle f_v/v'\rangle \wedge \psi\langle f_v/v\rangle$. If ϕ and ψ are transition formulas, we can think of this formula as representing a succession of two transitions, the first satisfying ϕ and the second satisfying ψ, with f representing the intermediate state. If we omit the subscript f, then the intention is that f is some set of variables not previously used. One important fact we will use is that $\{\phi\}t\{\psi\}$ is valid exactly when $\phi \wedge (t \times \neg\psi)$ is unsatisfiable.

Symbolic Interpreters. A *symbolic data state* represents a set of data states parametrically. The symbolic data states $\mathcal{S}$ consist of the triples (P, C, E), where P is a set of parameters (variables not in S), $C \in \mathcal{L}(P)$ is a constraint over the parameters, and E is a map from the program variables S to functions over P. We assume the these functions are expressible as first order terms over P. Thus, a symbolic state s can be characterized by the predicate $\chi(s) = C \wedge \bigwedge_{v \in S} v = E(v)$. A symbolic data state s represents a set of data states $\gamma(s)$ defined as follows:

$$\gamma(s) = \{d \in \mathcal{D} \mid d \models \exists P.\ \chi(s)\}$$

This is the set of data states produced by the map E for some valuation of the parameters satisfying the constraint C. We assume a defined initial symbolic data state s_o such that $\gamma(s_0) = \{d_o\}$. A (full) *symbolic state* is a pair $(l, s) \in \Lambda \times \mathcal{S}$.

A *symbolic interpreter* SI maps $\mathcal{A}$ to $\mathcal{S} \times \mathcal{S}$. We require that $\mathrm{SI}(a)$ is total for all actions a. Intuitively, a symbolic interpreter takes a symbolic state and an action, and returns a non-empty set of symbolic states representing the effect of executing a. Symbolic interpreter SI is *sound* when, for all symbolic states s and actions a,

$$\cup\gamma(\mathrm{SI}(a)(s)) = \mathrm{Sem}(a)(\gamma(s))$$

That is, the symbolic successors of s must together represent exactly the successors of the concrete states represented by s. Note that the set of states represented can be empty, since the constraint in a symbolic state can be F.

Note that $\mathrm{SI}(a)$ may be a function (i.e., deterministic). In this case, nondeterminism in a is modeled by the introduction of parameters. On the other hand, we may decide based on heuristic considerations to replace parameters with concrete values, introducing non-determinism in $\mathrm{SI}(a)$. Injecting concrete values in this way is analogous to decision making in DPLL. Soundness is not sacrificed as long as SI is sound. As in DART, however, it is also possible to substitute concrete values in an unsound way for operations which cannot be modeled symbolically [3].

2.1 Intraprocedural Algorithm

We first consider the case without procedure calls. We define a set of goals $G_0 \subset \Lambda$ that we wish to reach. For each goal, we wish to find a concrete run reaching the goal, or a proof that it is unreachable. The state of the algorithm is a triple (Q, A, G), where Q is a query set, A is a program annotation, and $G \subseteq \Lambda$ is the set of remaining goals. A *query* is a pair (s, ψ), where s is a symbolic state called the initial state, and ψ is a formula called the post-condition of the query. In the intraprocedural setting, the post-condition serves no purpose. It will be used later when computing procedure summaries.

An *annotation* is a set of pairs in $(\Lambda \cup \Delta) \times \mathcal{L}(S)$. We will notate these pairs in the form $l : \phi$ or $e : \phi$, where l is a location, e an edge and ϕ a formula called the *label*. The intended semantics is that no path beginning with location l or edge e can reach any remaining goal *if ϕ is initially true*. We will write $A(l)$ for $\vee\{\phi \mid l : \phi \in A\}$.

For an edge $e = (l_1, a, l_2)$, we say that a label $e : \phi$ is *justified* in A when $\{\phi\}\ a\ \{A(l_2)\}$, that is, when it implies the annotation of l_2 after executing a. We notate this condition $\mathcal{J}(e : \phi, A)$. For a location l, we will say that a label $l : \phi$ is justified in A when, for all edges $e \in \mathrm{Out}(l)$, there exists $e : \psi \in A$ such that $\phi \Rightarrow \psi$. An annotation is justified when all its elements are justified. A justified annotation is inductive. If it is also initially true, then it is an inductive invariant. Our algorithm maintains the invariant that A is always justified.

We will say that a query $q = ((l, s), \psi)$ is *blocked* by formula ϕ, when $s \models \phi$ and write $\mathcal{B}(q, \phi)$. With respect to q, the edge e is blocked when $\mathcal{B}(q, A(e))$, and the location l is blocked when $\mathcal{B}(q, A(l))$.

The algorithm IntraLA proceeds according to the transition rules defined in Figure 2. The initialization rule Init sets the algorithm state to $Q = \{((l_0, s_0), \psi_0)\}$, $A = A_0 = \emptyset$, $G = G_0$. That is, we are at the program's initial state, with no locations labeled, and all goals yet to be reached. The decision rule Decide generates a new query from an existing one by symbolically executing one program action. It may choose any edge that is not blocked, and any symbolic successor state generated by the action a. If the newly generated query is itself not blocked, it is added to the query set.

$$\textsc{Init}\ \frac{}{\{((l_0, s_0), \psi_0\},\ A_0,\ G_0}$$

$$\textsc{Decide}\ \frac{Q,\ A,\ G}{Q + ((l_2, s_2), \psi),\ A,\ G} \quad \begin{array}{l} q = ((l_1, s_1), \psi) \in Q \\ e = (l_1, a, l_2) \in \Delta \\ \neg\mathcal{B}(q, A(e)) \\ s_2 \in \mathrm{SI}(a)(s_1) \\ \neg\mathcal{B}(((l_2, s_2), \psi), A(l_2)) \end{array}$$

$$\textsc{Conjoin}\ \frac{Q, A, G}{Q - q,\ A + l : \phi,\ G - l} \quad \begin{array}{l} q = ((l, s), \psi) \in Q \\ \neg\mathcal{B}(q, A(l)) \\ \forall e \in \mathrm{Out}(l), e : \phi_e \in A \wedge \mathcal{B}(q, \phi_e) \\ \phi = \wedge\{\phi_e \mid e \in \mathrm{Out}(l)\} \end{array}$$

$$\textsc{Learn}\ \frac{Q, A, G}{Q,\ A + e : \phi,\ G} \quad \begin{array}{l} q = ((l_1, s_1), \psi) \in Q \\ e = (l_1, a, l_2) \in \Delta \\ \mathcal{B}(q, \phi) \\ \mathcal{J}(e : \phi, A) \end{array}$$

Fig. 2. Algorithm IntraLA

If all of the outgoing edges of a query are blocked, the Conjoin rule is used to block the query by labeling its location with the conjunction of the labels that block the outgoing edges. At this point, we know that the symbolic state is not empty (since otherwise the query would already be blocked). Thus, if the location is a goal, we have reached the goal, and we remove it from the set of remaining goals. The blocked query is discarded.

The remaining case is that some outgoing edge $e = (l_1, a, l_2)$ is not blocked, but every possible symbolic step along that edge leads to a blocked state. In this case, the Learn rule infers a new label ϕ that blocks the edge. The new edge label can be any formula ϕ that both blocks the current query and is justified. Such a formula can be obtained as an interpolant for (A, B), where $A = \chi(s_1)$ and $B = \mathrm{Sem}(a) \wedge \neg A(l_2)'$. Thus we can derive ϕ, if it exists, using an interpolating theorem prover [7].

The algorithm maintains the invariant that no queries are blocked, and, for every $l \in G$, $A(l) = F$. It terminates when no rules can be applied, which implies the query set is empty.

Theorem 1. *When algorithm* IntraLA *terminates, all the locations in* $G_0 \setminus G$ *are reachable and all the locations in* G *are unreachable.*

Proof sketch. All the rules preserve the invariant that A is justified (therefore inductive), that all the locations in G are unlabeled (meaning their annotation is equivalent to F) and that no queries are blocked. Now suppose the algorithm is in a state where no rules can be applied and consider some $q \in Q$. Since Decide does not apply, all possible successor queries are blocked. Thus, since Learn does

not apply, all outgoing edges are blocked. Thus, since CONJOIN does not apply, q is blocked, a contradiction. Since Q is empty, it follows that the initial query is blocked, meaning that $d_0 \models A(l_0)$. Therefore, A is an inductive invariant. Since all remaining goals are annotated F, it follows that they are unreachable. Moreover, since goals are only removed from G when reached, all locations in $G_0 \setminus G$ are reachable. □

Of course, we can also terminate the algorithm immediately if the set of remaining goals becomes empty.

2.2 Handling Unbounded Executions

The approach described above has one clear drawback: if the program has any loop that can execute unboundedly, then the algorithm will not terminate. That is, for any learning to occur, we must first reach a blocked state. However, if there is an unbounded loop, we can keep extending the run infinitely without reaching a blocked state.

To deal with this situation, we use the following generic approach. We introduce an auxiliary variable τ to the program. This variable must be non-increasing and infinitely often decreasing according to some pre-order that is well-founded over a domain characterized by some predicate W. For example, τ could be an integer that is decremented by every program action and the domain predicate could be $\tau >= 0$. We can think of τ as the program's "time to live". Alternatively, it would be sufficient to decrement τ on each back-edge of the program graph, so that τ has to be decremented at least once on each cycle. Or, τ could be a vector with one element for each SCC of the graph.

Now fix an initial value τ_0 of τ, and label every location l with the predicate $\neg W$. With this construction, every infinite run is eventually blocked. Thus, algorithm INTRALA is guaranteed to terminate (at least if the symbolic interpreter SI is finitely non-deterministic). When termination occurs, the annotation A is a proof that the remaining goals cannot be reached for the particular initial value τ_0. This is, in effect, a form of bounded model checking.

To obtain an unbounded proof, we can use the heuristic that bounded proofs may contain the ingredients of unbounded proofs. To do this, we will eliminate the dependence on τ in the annotation A, resulting in an unbounded annotation A_U. This can be done, for example by substituting some fixed value $\perp$ for τ, typically the bottom value of the pre-order.

The unbounded annotation A_U is not necessarily justified. However, we can make it justified by iteratively dropping labels that are not justified until a fixed point is reached. The result is the greatest inductive subset of A_U. This set of unbounded facts can then be used to strengthen A. If the resulting annotation is true in the initial state for all values of τ, then we have proved unreachability of the remaining goals. Otherwise, we increase τ_0 and repeat algorithm INTRALA. This overall procedure is depicted in Figure 3.

Because the problem of determining the reachable goals is undecidable, we do not expect this algorithm to terminate in all cases. The hope is that the

Algorithm StrengthenInput: a query q and goal set G_0
Output: set of unreachable goals

$Q \leftarrow \{q\}$, $A \leftarrow \emptyset$, $G \leftarrow G_0$
while T **do**
  Run IntraLA on (q, A, G) to termination
  $A_U \leftarrow \{l : \phi\langle\bot/\tau\rangle \mid l : \phi \in A\}$
  while exists $l : \phi \in A_U$ s. t. $\neg\mathcal{J}(l : \phi, A)$ **do**
    $A_U \leftarrow A_U - l : \phi$
  done
  if $\mathcal{B}(q, A_U(l_0))$ return G
  $A \leftarrow A \cup A_U$
  increase τ_0
done

Fig. 3. Algorithm with inductive strengthening

computed interpolants will converge to inductive assertions after a small number of iterations of the loops, and thus τ_0 will not become large. An alternative inductive strengthening approach would be to apply predicate abstraction using the atomic predicates occurring in A_U. Though the cost would be higher, the chance of convergence might be better.

We must also take care to handle loops with large fixed bounds efficiently. That is, suppose we have a loop that iterates N times where N is a large fixed number. If we increment τ_0 by one, then we may increment τ_0 N times before exiting the loop, resulting in $O(N^2)$ decision steps. One simple way to prevent this would be to double τ_0 instead of incrementing it, which would give $O(N \log N)$ steps. Alternatively, if we can determine statically that the loop is bounded, we can simply remove the decrements of τ from the loop, without causing non-termination of IntraLA.

2.3 Interprocedural Algorithm

To handle procedures in a modular fashion, we will annotate the program with *negative summaries*. This is a transition formula ϕ with the intended meaning that if $\phi[d_0, d_1]$ holds, then entry to the procedure in data state d_0 may *not* result in exit in state d_1. Negative summaries are used because they are inductive in the normal, forward sense.

To detect goals reached within procedures, we designate a special variable f. On reaching a goal, a procedure *aborts*, that is, it exits immediately with f true. On normal exit, f is false. Given a negative summary ϕ, we can think of $\phi\langle T/f'\rangle$ as a pre-condition under which the procedure guarantees not to reach a goal and abort.

Modeling Programs with Procedures. To model data in programs with procedures we designate a set of global variables G and local variables L. For $i = 0, 1, \ldots$ the *frame* L_i consists of a variable v_i for each $v \in L$. A frame

Function SUM(q_0, A, G)

Apply INTRALA from initial state $(\{q_0\}, A, G)$ until
1) there exists $((l, d), \psi) \in Q$, where $l \in \Omega$:
in which case, return (d, A, G), *or*
2) Q is empty:
in which case, return (ϵ, A, G).

Fig. 4. Algorithm for procedure summary construction

represents the local state of one procedure instance, with L_0 representing the current procedure instance.

To model procedure calls, we introduce special actions call(l), where l is a location of a procedure to be called. We introduce a set of procedure call edges Θ that are distinct from the ordinary edges Δ. We also designate a set $\Omega \subset \Lambda$ of *exit states*. Due to space considerations, we give the semantics of calls only informally. The effect of an edge $(l_1, \text{call}(l_2), l_3)$ is to transfer control to l_2, execute until reaching an exit state, then return by transferring control to l_3. The effect of a call action on the data state is to "push" one frame on the "stack". We define an operation push on data states that shifts the local variables up by one frame, so that $\text{push}(d)(v_{i+1}) = d(v_i)$. On return, one stack frame is "popped". We define $\text{pop}(d)$ so that $\text{pop}(d)(v_i) = d(v_{i+1})$. We also define corresponding renaming operators on formulas, so that $\text{push}(\phi) = \phi\langle L_{i+1}/L_i\rangle$ and $\text{pop}(\phi) = \phi\langle L_{i-1}/L_i\rangle$.

Computing summaries. We will say a query $q = ((l, s), \psi)$ is *blocked* by a negative summary ϕ, that is, $\mathcal{B}(q, \phi)$, when every exit allowed by ϕ satisfies the post-condition ψ. Because summaries are negative, this is equivalent to saying that $\{\chi(s)\}\ \neg\phi\ \{\psi\}$ or that the formula $\chi(s) \wedge (\neg\phi \times \neg\psi)$ is unsatisfiable.

Justification of ordinary edges and locations remains as before. Now, however, we say that ϕ is justified at an *exit state* l when $\phi = \neg I_S \vee f'$, where I_S is the identity relation over the data variables S. In other words, exiting from a procedure leaves the data unchanged, and does *not* abort. We will consider justification of call edges shortly.

Using algorithm INTRALA, we can define a function SUM (see Figure 4) that constructs a negative summary for a procedure. It takes a query $((l_1, d_1), \psi)$ and returns either a reachable exit state d_2 that does *not* satisfy the post-condition ψ (*i.e.* a counterexample to the query) or it labels l_1 with a summary that ensures that no such counterexample exists.

Using procedure summaries. We use procedure summaries to justify the annotation of call edges. Intuitively, a summary ϕ is justified at a call edge $e = (l_1, \text{call}(l_2), l_3)$ if it is justified by considering the summary of the called procedure l_2 as an action. There are two subtleties involved in this, however. The first is that we must account for the shift in stack frames between the calling and called contexts. We can do this by applying the push operator to the formulas in the calling context. The second is that the calling context must abort if the called context aborts. We can effect this by weakening the summary at the return

location. We define $\mathcal{W}(\phi) = \phi \wedge \neg(f \wedge f')$. In effect, this removes transitions from the negative summary at the return site, allowing the calling context to abort when the called function does. Using this notion, the justification condition for procedure calls is defined as follows:

$$\mathcal{J}((l_1, \text{call}(l_2), l_3) : \phi, A) \quad \text{iff} \quad \{\text{push}(\phi)\} \;\; \neg A(l_2) \;\; \{\text{push}(\mathcal{W}(A(l_3))\}$$

We can then prove the following lemma:

Lemma 1. *If negative summary annotation A is justified, and if $d_0 \models A(l_0)\langle T/f' \rangle$ and if $A(l) = F$ for some location l, then location l is unreachable.*

Proof sketch. By induction on the length of runs, we show that if the initial state (l_0, d_0) of a run satisfies $A(l_0)\langle T/f' \rangle$ then every state (l_i, d_i) satisfies $A(l_i)\langle T/f' \rangle$. This property is preserved by call actions because we have weakened the summary at the return site, so that the caller aborts when the callee aborts. As a result, no reachable state can be labeled F. □

Now we are ready to define versions of the DECIDE and LEARN rules for call edges. These are shown in Figure 5. In both cases, we have an outgoing call edge e from the current query. We compute a post-condition ψ_2 for the called function based on the current summary of the return site and the post-condition of the calling context. Notice we use the weakened summary to allow the called function to abort. Also notice that we apply push to the current state when entering the called function. If SUM returns an exit state, then the return site is not blocked, and DECIDEC generates a new query after the call (note pop is applied to this state). On the other hand, if SUM returns ϵ, then we are blocked. Thus LEARNC can annotate the edge with a blocking formula ϕ.

A suitable condition ϕ that is both blocking and justified can be obtained as an interpolant for (A, B), where $A = \chi(s_1) \wedge \neg\psi_1'$ and $B = \neg\text{pop}(A(l_2)) \times \neg\mathcal{W}(A(l_3))$. That is, the transition formulas implied by A are exactly the negative summaries blocking the query q (note s_1 is the symbolic state of q and ψ_1 is the post-condition of q). The transition formulas inconsistent with B are exactly those justified at the call edge. Moreover, an interpolant for (A, B) must be a transition formula, since only variables in $S \cup S'$ can be common to A and B. Thus, any interpolant for (A, B) satisfies the conditions for the annotation ϕ of the call edge.

We will call the algorithm INTRALA with the addition of these two rules INTERLA. The initial annotation A_0 consists of the labels $r : \neg I_S \vee f'$, for all the exit locations $r \in \Omega$. The initial post-condition $\psi_0 = \neg f$. Essentially, this constructs a summary that proves that the program does not abort. This procedure is recursive. When it encounters a call edge, it calls SUM with a suitable query. This in turn runs INTERLA on the called procedure. The recursive call can result in the addition of labels, and the elimination of reachable goals.

Theorem 2. *When algorithm* INTERLA *terminates, all the locations in $G_0 \setminus G$ are reachable and all the locations in G are unreachable.*

$$\textsc{DecideC}\frac{Q,\ A,\ G}{Q + ((l_3, \mathrm{pop}(s_3)), \psi_1),\ A_3,\ G_3} \quad \begin{array}{l}(s_3, A_3, G_3) = \textsc{Sum}(q_2, A, G)\\ \neg\mathcal{B}(q, A(e))\end{array}$$

$$\textsc{LearnC}\frac{Q, A, G}{Q,\ A + e : \phi,\ G} \quad \begin{array}{l}(\epsilon, A_3, G_3) = \textsc{Sum}(q_2, A, G)\\ \mathcal{B}(q, \phi)\\ \mathcal{J}(e : \phi, A_3)\end{array}$$

$$\text{where} \quad \begin{cases} q = ((l_1, s_1), \psi_1) \in Q \\ e = (l_1, \mathrm{call}(l_2), l_3) \in \Theta \\ \psi_2 = \neg\mathcal{W}(A(l_3)) \times \psi_1 \\ q_2 = ((l_2, \mathrm{push}(s_1), \psi_2) \end{cases}$$

Fig. 5. Rules for InterLA (in addition to IntraLA)

Proof sketch. As in the intraprocedural case, all the rules preserve the invariant that A is justified and that the locations in G are unlabeled. When the initial query $((l_0, s_0), \neg f)$ is blocked, we know that $d_0 \models A(l_0)\langle T/f'\rangle$ therefore by the lemma, all unlabeled locations are unreachable. □

3 Implementation and Experiments

Algorithm InterLA has been implemented in a tool we will call Impact II. This tool uses the LLVM compiler infrastructure [6] to generate CFG's from C programs, with basic blocks corresponding to edges of the graph. The tool uses the FOCI interpolating prover [7] both for checking satisfiability of formulas and computing interpolants. The C heap is modeled using the theory of arrays. Static analysis is used to partition loads and stores into alias classes, with each alias class modeled by an array. External functions are modeled as having no side effects and returning a non-deterministic value, which can be considered an input to the program.

When a goal location is reached, Impact II extracts a satisfying assignment to the symbolic constraint C of the symbolic state as a test case. By initially marking every location as a goal, we can generate a set of tests that provides 100% coverage of reachable locations, and prove that the uncovered locations are unreachable (since the compiler inlines small functions, some goals may be duplicated).

Experiments were conducted to test the hypothesis that this approach can produce a greater diversity of program behavior more quickly than methods such as DART that enumerate all execution paths. We used as benchmarks several device driver examples previously used as software model checking benchmarks [8]. We compare InterLA against an implementation of DART using the same symbolic interpreter and prover. This allows us to gauge the effect of learned annotations in guiding the search. Without learning, we simply enumerate all possible control paths. DART terminates in these tests because all the loops are bounded and there is no recursion.

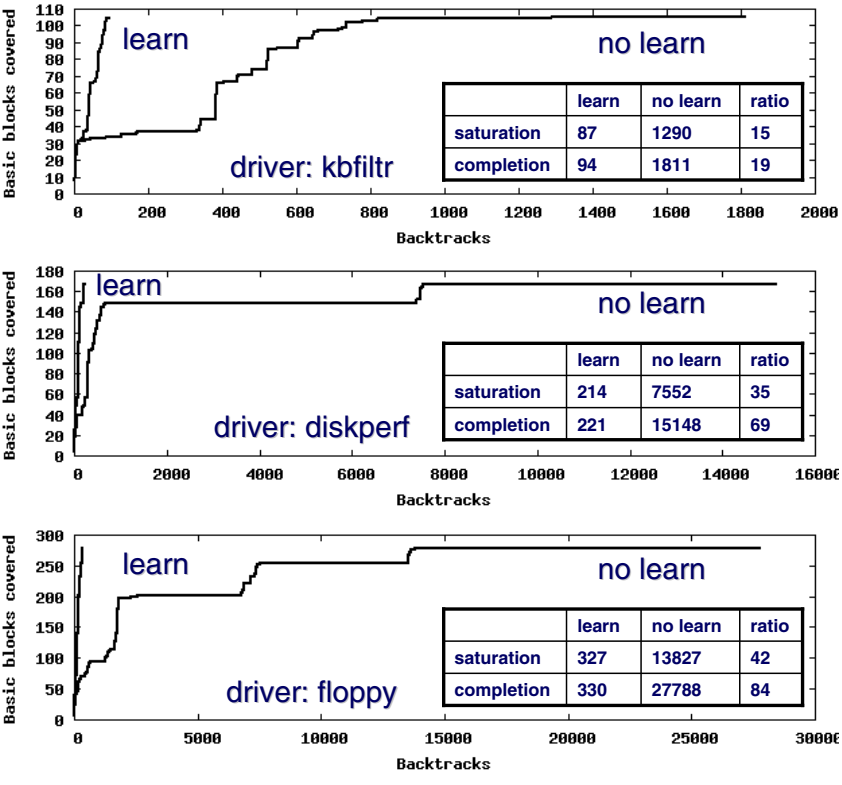


Fig. 6. Comparison of test generation with and without learning

Figure 6 plots the number of coverage goals reached as a function of the number of times the symbolic interpreter backtracked to an alternative program branch (which is also the number of test sequences generated). Three examples[1] are shown, with the number of reachable basic blocks ranging from 104 to 279. Each plot shows a line for InterLA ("learn") and a line for DART ("no learn"). The tables compare the number of backtracks needed for saturation (all reachable locations reached) and completion. Without learning, there are long plateaus during which many paths are explored but no new locations are reached. Learning clearly acts to push the search away from these regions, allowing the search to make steady progress. This effect is more pronounced in the larger program, with learning reducing backtracks to completion by a factor 84.

4 Conclusion

Lazy annotation allows us to deduce program annotations in response to search failure, much in the way that a DPLL SAT solver learns conflict clauses. As we

[1] Source code available at `http://www.kenmcmil.com/benchmarks.html`

have seen, this allows us to prune the search in test generation to achieve a given coverage goal at a greatly reduced cost. The method also has some potential advantages with respect to existing software model checking techniques. Since it is based entirely on interpolants, it avoids the expense of quantifier elimination or predicate image computations. The annotation approach avoids irreversible partitioning of the abstract state space, and potentially allows more general procedure summaries. Compared to lazy abstraction with interpolants, the method allows procedure summarization (thus may be more effective for deeply nested procedures) and handles bounded loops more effectively. In the other hand, it may be that the lazy abstraction approach of exploring infeasible program paths produces interpolants that are more relevant to the property being checked. Moreover, the question of how to obtain convergence in practice for unbounded loops needs further study.

References

1. Ball, T., Rajamani, S.K.: The SLAM project: Debugging system software via static analysis. In: POPL, pp. 1–3 (2002)
2. Craig, W.: Three uses of the Herbrand-Gentzen theorem in relating model theory and proof theory. J. Symbolic Logic 22(3), 269–285 (1957)
3. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: PLDI, pp. 213–223 (2005)
4. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL, pp. 43–56 (2010)
5. Gulavani, B., Henzinger, T.A., Kannan, Y., Nori, A., Rajamani, S.K.: Synergy: A new algorithm for property checking. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 117–127. Springer, Heidelberg (2006)
6. Lattner, C., Adve, V.: LLVM: A compilation framework for lifelong program analysis & transformation. In: CGO, pp. 75–88 (2004)
7. McMillan, K.L.: An interpolating theorem prover. Theor. Comput. Sci. 345(1), 101–121 (2005)
8. 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)
9. Majumdar, R., Henzinger, T.A., Jhala, R., Sutre, G.: Lazy abstraction. In: POPL, pp. 58–70 (2002)

The Static Driver Verifier Research Platform

Thomas Ball, Ella Bounimova, Vladimir Levin
Rahul Kumar, and Jakob Lichtenberg

Microsoft Corporation
http://research.microsoft.com/slam/

Abstract. The SDV Research Platform (SDVRP) is a new academic release of Static Driver Verifier (SDV) and the SLAM software model checker that contains: (1) a parameterized version of SDV that allows one to write custom API rules for APIs independent of device drivers; (2) thousands of Boolean programs generated by SDV in the course of verifying Windows device drivers, including the functional and performance results (of the BEBOP model checker) and test scripts to allow comparison against other Boolean program model checkers; (3) a new version of the SLAM analysis engine, called SLAM2, that is much more robust and performant.

1 Introduction

Static Driver Verifier [1] (SDV) is a verification tool included in the Windows Driver Kit (WDK), using SLAM [4] as the underlying analysis engine. SDV comes with support for three classes of drivers: WDM (The Windows Driver Model); KMDF (Kernel Mode Driver Framework); NDIS (Network Driver Interface Specification). For each of these driver classes, SDV provides a number of class-specific components (for example, API rules and an environment model). API rules are expressed in the SLIC language [5] and describe the proper way to use the driver APIs.

The SDV Research Platform (SDVRP) is a new academic release of SDV that contains a number of features that should be useful to the verification research community:

- **Static Module Verification:** SDVRP enables the development of SLIC rules for APIs independent of device drivers, and the application of SDV to modules that use these APIs. With this feature, researchers can use SDV to verify that clients of an API adhere to the API specification.
- **Boolean Program Repository and Test Scripts:** SDVRP contains thousands of Boolean programs generated by SDV in the course of verifying Windows device drivers, including the functional and performance results of running the symbolic model checker BEBOP [3] on these programs.
- **Slam2 Engine:** SDVRP contains a new version of the SLAM analysis engine (SLAM2) that is much more robust and performant than the first version of SLAM. SDV for Windows 7 uses SLAM2.

SDVRP is available from `http://research.microsoft.com/slam/` under a license for academic use.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 119–122, 2010.

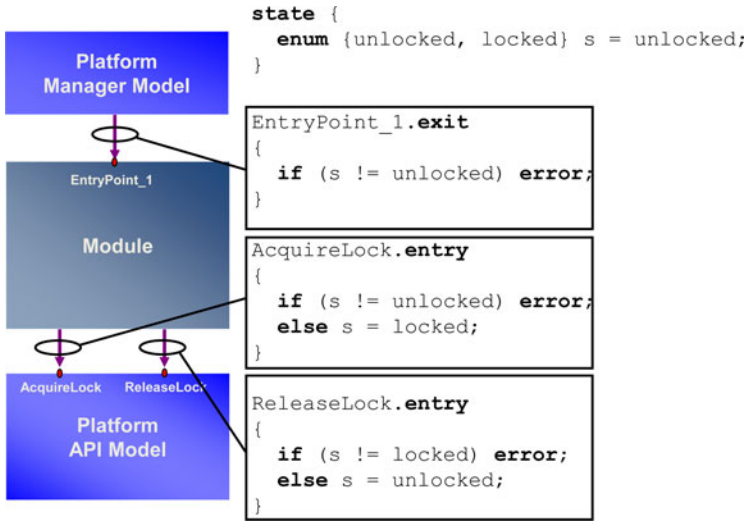


Fig. 1. The interaction between the platform model, a module, and a rule. The platform manager model calls into the entry points of the module. The module itself interacts with the underlying API model, while the rule specifies the safe interactions between the various components.

2 Static Module Verification

In its early days, SDV verified API usage requirements on WDM Drivers. With the success of SDV, came the difficulty of scaling out to other classes of drivers and programs, which in turn motivated the need to parameterize SDV so it could be adapted for other uses.

At a high level, SDV consists of a verification engine, a model of the operating system (platform/environment model), and a set of driver API rules. The verification engine checks whether a user provided driver in the context of the operating system model adheres to the applicable driver API usage rules.

SDVRP generalizes this concept by allowing researchers to provide their own version of the platform model and the API usage rules. Together these two parts comprise a *plugin for static module verification*. The verification engine now checks whether a user provided C module adheres to the plugin API usage rules, in the context of the plugin platform model. This allows SDV to be applied to many other pieces of code besides device drivers.

The platform model itself can be thought of as having two major parts. First, the platform model implements how the platform exercises the module by calling into the module's entry points. This is done by the platform manager model. We can think of this component as the "main" routine of the system. Second, the platform API model provides an implementation of the APIs that the module can use. They are simplified implementations of each platform API that contain behaviors relevant for the verification of associated platform API rules. The

platform model is written using the C language, with one special construct for introducing non-deterministic choice.

Figure 1 shows the interaction between the platform model, the module, and a rule. The platform manager model calls into the entry points of the module. The module itself interacts with the underlying API model, while the rule monitors the interactions between the various components. SLIC [5] rules allow declaration of state as well as state transitions based on API events (call/return). When SDV finds a rule violation, it constructs an error trace that passes through the platform model, the module, and the rule.

Along with the three highly developed plugins for existing driver platform models, SDVRP also comes with a minimal plugin. All of these are available for use, modification, and cloning for research purposes.

3 Boolean Program Repository

SDV/SLAM generates Boolean programs that represent abstractions of C programs, where each Boolean variable represents a predicate on the state of the C program. Boolean programs are an interesting object of study because they admit efficient symbolic model checking, despite the fact that they have recursive procedures. BEBOP [3] is SLAM's model checker for Boolean programs. SDV runs on many drivers, for each driver checking many SLIC rules. A single run of SDV on a driver against a rule can generate many Boolean programs, one for each iteration of the counterexample guided abstraction refinement (CEGAR) process, which successively refines the Boolean program. The SDVRP contains the Boolean programs generated by SDV when run on the drivers in the WDK, as well as the functional and performance results of running BEBOP on these programs. Furthermore, the SDVRP contains the set of test scripts used to generate the results, so that others may easily substitute other Boolean program model checkers in place of BEBOP.

4 Slam2 Engine

SLAM2 improves the precision, reliability, maintainability and scalability of the original SLAM verification engine (SLAM1). SDV 2.0, released with the Windows 7 WDK, uses SLAM2. For SDV 2.0, the true bugs/total bugs ratio is 90-98% on Windows 7 Microsoft drivers, depending on the class of driver. The number of non-useful results (timeouts, "don't know" results) has been reduced greatly. In particular, for drivers shipped as WDK samples, it is 3.5% for WDM drivers and 0.02% for KMDF drivers.

Comparing SLAM2 to SLAM1, on WDM drivers SLAM1 had 19.7% false defects (31/157 reported defects), while SLAM2 had 0.4% (2/512). On WDM drivers, SLAM1 had 6% "give-up" runs (285/4692), while SLAM2 had 3.2% (187/5727). On KMDF drivers SLAM1 had 25% false defects (75/300), while SLAM2 had 0% (0/271). On KMDF drivers SLAM1 had 1% "give-up" runs (31/3111), while SLAM2 had 0.004% (2/5202)

SLAM2 implements a CEGAR loop, which consists of the following main components: a predicate abstraction module, a model checker, and an error trace validation/predicate discovery module. SLAM2 has a new field-sensitive alias analysis with improved precision and performance, and uses the Z3 state-of-the-art SMT solver [6] and new axiomatization of pointer aliasing [2].

The changes in SLAM2 are mostly related to the abstraction, trace validation and predicate discovery functionalities of the CEGAR loop. An abstract intermediate representation (IR) of the input program is introduced as an interface between the low-level IR representing the C program and the CEGAR loop, which permits independence from the input language and from the granularity of abstraction (single statement or multiple statements).

The error trace validation algorithm in SLAM2 is bi-directional with respect to the error trace, combining forward symbolic execution of the error trace (strongest postconditions) with backwards symbolic execution (weakest preconditions). As a result, significant optimization is achieved on long error traces often encountered in the runs on Windows Device Drivers.

Forward execution computes data about the trace (procedure call graph, variable values at each step, pointer aliasing, etc.). The data is used to perform simple feasibility checks on the trace and to optimize subsequent backwards execution and predicate discovery algorithms. Backwards execution is optimized by taking into account data about the trace discovered on the forward pass (for example, program-point-specific pointer aliasing).

SLAM2 implements a new algorithm for discovering Boolean predicates, which is a part of the backwards execution pass. The algorithm is iterative and progresses (on an as-needed basic) from generating a small set of new predicates via computationally cheaper techniques, towards larger sets of predicates via more expensive discovery algorithms.

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys, pp. 73–85 (2006)
2. Ball, T., Bounimova, E., de Moura, L., Levin, V.: Efficient evaluation of pointer predicates with z3 smt solver in slam2. Technical Report MSR-TR-2010-24, Microsoft Research (2010)
3. Ball, T., Rajamani, S.K.: Bebop: A symbolic model checker for Boolean programs. In: Havelund, K., Penix, J., Visser, W. (eds.) SPIN 2000. LNCS, vol. 1885, pp. 113–130. Springer, Heidelberg (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. Ball, T., Rajamani, S.K.: SLIC: A specification language for interface checking. Technical Report MSR-TR-2001-21, Microsoft Research (2001)
6. 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)

Dsolve: Safety Verification via Liquid Types*

Ming Kawaguchi, Patrick M. Rondon, and Ranjit Jhala

University of California, San Diego
{mwookawa,prondon,jhala}@cs.ucsd.edu

Abstract. We present DSOLVE, a verification tool for OCAML. DSOLVE automates verification by inferring "Liquid" refinement types that are expressive enough to verify a variety of complex safety properties.

1 Overview

Refinement types are a means of expressing rich program invariants by combining classical types with logical predicates. For example, using refinement types, one can express the fact that `x` is an array of positive integers by stating that `x` has the type $\{\nu : \texttt{int} \mid 0 < \nu\}$ `array`. While refinement types have been shown to be a powerful technique for verifying higher-order functional programs [1–4], refinement type systems have previously been difficult to use because of a high programmer annotation burden.

We present DSOLVE, a tool that automates the verification of safety properties of OCAML programs by inferring refinement types. Using DSOLVE, we were able to verify properties of real-world OCAML programs as diverse as array bounds safety and correctness of sorting and tree-balancing algorithms while incurring a modest overhead in terms of the annotations and hints required for verification. Further, we were able to use the refinement types inferred by DSOLVE on buggy programs to diagnose and correct the problems, demonstrating its value as a tool for program understanding.

DSOLVE works by inferring *Liquid Types*, which are refinement types whose refinements are conjunctions of predicates taken from a user-provided finite set of *logical qualifiers*. Each logical qualifier is a predicate over the program variables and the special value variable ν, which is used to refer to values of the refined type. The Liquid Type restriction makes inference tractable while still retaining enough expressiveness to verify safety properties of real-world OCAML programs.

2 Example

In this section, we illustrate Liquid Types and show how DSOLVE is able to verify a polymorphic, higher-order, array-manipulating program, shown in Figure 1. We will show how DSOLVE statically verifies the safety of the program's array

* This work was supported by NSF grants CCF-0644361, CNS-0720802, CCF-0702603, and a gift from Microsoft Research.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 123–126, 2010.

```
1  let rec foldn m n b g =
2    if m < n then foldn (m+1) n (g m b) g else b
3
4  let weighted_avg x w =
5    if Array.length x > 0 && Array.length x = Array.length w then
6      let b = x.(0) * w.(0), w.(0) in
7      let f = fun i (sum, n) -> sum + x.(i) * w.(i), n + w.(i) in
8      let sum, n = foldn 1 (Array.length x) b f in
9        sum / n
10   else
11     assert false
12
13 let _ = weighted_avg [|10; 15; 20|] [|1; 1; 1|]
```

Fig. 1. An Example OCaml Program

accesses and division operation (i.e., that the array indices are within bounds on lines 5, 6, 7 and that the denominator is non-zero on line 9).

Qualifiers. Dsolve takes a set of logical qualifiers as input from the user, which it uses to construct refinement types. Assume that the user has supplied the following qualifiers: $\{0 < \nu,\ \star \leq \nu,\ \nu < \star,\ \nu < \texttt{len}\ \star\}$, where the uninterpreted function symbol `len` is an abbreviation for `Array.length` and $\star$ denotes a "wildcard" that is instantiated with program variables.

The higher-order function `foldn` folds over the integers from `m` to `n`. Dsolve infers that `foldn` calls `g` with values between `m` and `n`, that is, `foldn` has type

$$\texttt{m:int} \rightarrow \texttt{n:int} \rightarrow\ \alpha \rightarrow (\texttt{g:}\{\nu : \texttt{int} \mid \texttt{m} \leq \nu \wedge \nu < \texttt{n}\} \rightarrow \alpha \rightarrow \alpha) \rightarrow \alpha.$$

This is a Liquid Type since the refinement for the input of `g` is the conjunction of $\star \leq \nu$ and $\nu < \star$, where the wildcards are instantiated with `m` and `n`, respectively.

The function `weighted_avg` uses `foldn` to compute the weighted average of the array `x`'s values using the corresponding weights in array `w`. From the call on line 13, Dsolve infers that `x` and `w` have the same positive length and that `w` contains only positive entries. Dsolve then determines that the condition on line 5 is always true, so the assertion on line 11 never executes, and that the array accesses on line 6 are within bounds. Using the types of `foldn` and `x`, Dsolve also determines that function `f` on line 7 has type

$$\texttt{f} :: \{\nu : \texttt{int} \mid 0 < \nu \wedge \nu < \texttt{len x} \wedge \nu < \texttt{len w}\} \rightarrow \texttt{int} * \texttt{pos} \rightarrow \texttt{int} * \texttt{pos}.$$

where `pos` abbreviates $\{\nu : \texttt{int} \mid 0 < \nu\}$. Thus, Dsolve determines that all accesses to arrays `x` and `w` within `f` are safe. Finally, Dsolve determines from `f`'s type that `n` is always positive, and so the division on line 9 is safe.

Modular Verification. Dsolve verified the safety of this program using whole-program analysis, i.e., by analyzing the call to `weighted_avg` on line 13. The programmer could also verify the above program by writing the following interface specification (or "contract") for `weighted_avg`:

$$\texttt{x:}\{\nu : \texttt{int array} \mid \texttt{len}\ \nu > 0\} \rightarrow \{\nu : \texttt{pos array} \mid \texttt{len}\ \nu = \texttt{len x}\} \rightarrow \texttt{int}$$

Dsolve can use these specifications to verify modules without driver code and also to verify a module's clients.

3 Tool

Architecture. DSOLVE is divided into the following three phases, described in detail in [5, 6]. First, the OCAML compiler's parser and typechecker are used to translate the input program to a typed AST; this phase also parses the module's refinement type specification. Second, the typed AST is traversed to generate a set of subtyping constraints over templates that represent the potentially unknown refinement types of the program expressions. Third, the constraints are solved using predicate abstraction over a finite set of predicates generated from user-provided logical qualifiers. This pass uses the Z3 SMT solver [7] to discharge logical implications corresponding to the subtyping constraints. If the constraints can be satisfied, the program is deemed safe. Otherwise, DSOLVE reports a type error and the lines in the original source program that yielded the unsatisfiable constraints.

DSOLVE is conservative. If an error is reported, it may be because the program is unsafe, or because the set of qualifiers provided was insufficient, or because the invariants needed to prove safety cannot be expressed within our refinement type system.

Input. DSOLVE takes as input a source (`.ml`) file containing an OCAML program, an interface (`.mlq`) file containing a refinement type specification for the interface functions of the `.ml` file, and a qualifier (`.hquals`) file containing a set of logical qualifiers. DSOLVE combines the qualifiers from the `.hquals` file with some scraped from the specification `.mlq` file and a standard qualifier library to obtain the set of logical qualifiers used to infer liquid types.

Output. DSOLVE produces as output a refinement type for each program expression in a standard OCAML type annotation (`.annot`) file. The user can view the inferred refinement types using standard tools like EMACS, VIM, and CAML2HTML. If all the constraints are satisfied, the program is reported as safe. Otherwise, DSOLVE outputs warnings indicating the potentially unsafe expressions in the program.

Modular Checking. DSOLVE verifies one module at a time. If a module depends on another module, it can be checked against that module's `.mlq` file; the other module's source code is not required.

Abstract Modules. It is possible to create a `.mlq` file which defines types, axioms (background predicates), and uninterpreted functions without a `.ml` file. Such "abstract modules" allow the user to extend DSOLVE with reasoning about mathematical structures which do not appear directly in the program. For example, an abstract module `Set.mlq` might contain a type which represents a polymorphic set collection, along with an appropriate refined interface and axioms which build a set theory. This set theory can be used in another module's type refinements; for example, it may be used in a sorting module to verify that the sets of elements in the input and output lists of a sorting function are equal.

Availability. The DSOLVE source distribution is available, along with benchmarks and an online demo, at `http://pho.ucsd.edu/liquid/`.

4 Experiments

We report the results of applying DSOLVE to real-world OCAML programs.

Static Array Bounds Checking. We have previously used benchmarks from the DML project [1, 8] to show that DSOLVE significantly reduces annotation overhead burden in the static verification of array safety [5]. In our study, we automatically generated qualifiers of the form $\nu \bowtie X$, where $\bowtie \in \{<, \leq, =, \neq, >, \geq\}$ and $X \in \{0, \star, \texttt{len}\ \star\}$. This allowed us to reduce annotation overhead from 17% of LOC using DML to under 1% of LOC using DSOLVE. Runtimes ranged from 1 to 64 seconds, the longest being for BITV [9], a 426-line bit vector library.

Data Structures. We have also used DSOLVE to verify data structure invariants in production OCAML libraries [6], including that OCAML's `List.stablesort` outputs a sorted list, that OCAML's `Map` module implements an AVL tree and that `Map`'s keys form a set. Runtimes in this study ranged from 1 to 103 seconds, the longest being for VEC [10], a 343-line OCAML extensible array library.

Program Understanding. DSOLVE also helped us find and fix a subtle bug in VEC. A VEC extensible array is represented by a balanced tree with a balance factor of at most 2. As originally released, VEC contained a flawed recursive balancing routine, `recbal`, which was meant to efficiently merge two balanced trees of arbitrarily different heights into a single balanced tree. When run on this code, the strongest invariant DSOLVE could infer was that the output tree would have a balance factor of at most 4. By changing `recbal` and re-inferring types, we were able to isolate the faulty code paths and find test inputs with output balance factor of 4. DSOLVE verified the fix, which the author adopted.

References

1. Xi, H., Pfenning, F.: Eliminating array bound checking through dependent types. In: PLDI (1998)
2. Cui, S., Donnelly, K., Xi, H.: ATS: A language that combines programming with theorem proving. In: Gramlich, B. (ed.) FroCos 2005. LNCS (LNAI), vol. 3717, pp. 310–320. Springer, Heidelberg (2005)
3. Bengtson, J., Bhargavan, K., Fournet, C., Gordon, A.D., Maffeis, S.: Refinement types for secure implementations. In: CSF (2008)
4. Dunfield, J.: A Unified System of Type Refinements. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA (2007)
5. Rondon, P., Kawaguchi, M., Jhala, R.: Liquid types. In: PLDI (2008)
6. Kawaguchi, M., Rondon, P., Jhala, R.: Type-based data structure verification. In: PLDI, pp. 304–315 (2009)
7. 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)
8. Xi, H.: DML code examples, `http://www.cs.bu.edu/fac/hwxi/DML/`
9. Filliátre, J.C.: Bitv, `http://www.lri.fr/~filliatr/software.en.html`
10. de Alfaro, L.: Vec, `http://www.dealfaro.com/~luca/vec.html`

CONTESSA: Concurrency Testing Augmented with Symbolic Analysis

Sudipta Kundu*, Malay K. Ganai, and Chao Wang

NEC Labs America, Princeton, NJ, USA

Abstract. Testing of multi-threaded programs poses enormous challenges. To improve the coverage of testing, we present a framework named CONTESSA that augments conventional testing (concrete execution) with symbolic analysis in a scalable and efficient manner to explore both thread interleaving and input data space. It is built on partial-order reduction techniques that generate verification conditions with reduced size and search space. It also provides a visual support for debugging the witness traces. We show its significance in testbeds.

1 Introduction

Concurrency testing poses a major challenge due to large *interleaving* space of concurrent programs. To expose a concurrency bug, a test case should not only provide a bug-exposing input, but also provide a bug-triggering execution interleaving. Testing a program's behavior for every interleaving on every test input is infeasible.

Dynamic model checking [1–3], for a given test input performs systematic execution of a program under different thread interleavings. Even for a fixed test input, explicit enumeration of interleavings can still be quite expensive. Although partial order reduction techniques (POR) reduce the set of interleavings to explore, the reduced set often remains prohibitively large. Some previous work use ad-hoc approaches such as perturbing program execution by injecting artificial delays after every synchronization points [4], or use randomized dynamic analysis to detect real races [5]. Although such approaches addresses scalability, often they do not provide adequate coverage.

1.1 CONTESSA Framework: Overview

To improve the coverage of testing, we present a framework named CONTESSA that augments conventional CONcurrency TESting with Symbolic Analysis in a scalable and efficient manner to explore both thread interleavings *and* the input data space, as shown in the Figure 1.

First we automatically instrument a given source code (a multi-threaded C/C++ program) for logging global access events. We then obtain an executable binary of the instrumented code. (Alternately, one can instrument the binary directly for logging the global events.) We run the binary on a given set of test cases that include monitors corresponding to reachability properties such as common program errors, data races,

* Sudipta Kundu worked on the project as an intern at NECLA. He is now at Synopsys Inc.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 127–131, 2010.

atomicity violations [6]. Corresponding to each run, we obtain a corresponding concrete execution trace. From a set of these traces, we derive a lean partition of the program called a concurrent trace program (CTP) [7]. Implicitly, such a CTP captures all linearizations of the trace events that respect the control flow of the program.

To strike a balance between coverage and scalability, we use such a derived CTP to drive our symbolic analysis, i.e., to explore various interleavings symbolically to validate a given set of reachability properties. Specifically, the search engine combines a partial-order reduction technique [8] with a token-based (asynchronous) modeling approach [9] to generate verification conditions directly without an explicit scheduler. The corresponding formula is efficiently encoded [8] to obtain reduction both in the size and the interleaving search space. Such formulas can then be solved by the state-of-the-art SMT (Satisfiability Modulo Theory) solvers (e.g. [10]) with relative ease. The witness traces corresponding to the properties can be visualized in an event trace viewer, making debugging process easier. The tool currently supports C/C++ programs on the Linux/Pthreads platforms.

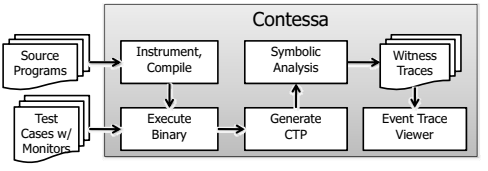


Fig. 1. Concurrent testing framework

Although one can derive the verification conditions directly from a source code (e.g. [9]), they are typically modeled imprecisely due to dynamic data elements such as pointers, linked lists, arrays and library calls. Such imprecision typically leads to spurious witnesses. To overcome the issue of spuriousness, we generate these conditions directly from CTPs which includes all the valid program traces. As a CTP is much smaller in size compared to the entire source program, it leads to manageable-sized verification conditions. Moreover, such CTPs can also be derived easily from prevalent testing infrastructures. In short, the strength of the tool is in finding "error traces" based on symbolic analysis of a set of "good traces."

2 Tool Flow

We highlight the various steps of the tool chain with an example shown in Figure 2. The example is a multi-threaded C program $\{$`foo, bar`$\}$ with shared variables G, H, L. The `test` harness invokes the main program `foo` with various `random` test values. The program is automatically instrumented (not shown) to log various memory accesses and synchronization events (denoted as t_i) during execution. The trace programs TP_α and TP_β correspond to two traces $\alpha = t_{10}\{t_{11} \cdots t_{17}\}\{t_{21} \cdots t_{25}\}t_{18}t_{19}$ with $x = 0, G = 1, H = 0$, and $\beta = t_{10}t_{11}\{t_{21} \cdots t_{24}\}t_{12}t_{13}t_{16}t_{17}t_{25}t_{18}t_{19}$ with $x = 3, G = 0, H = 0$, respectively of the test system. The concrete values of trace events are shown in the brackets (and underlined). Note, a trace program denotes a totally ordered events. The assertion at t_{19} denotes the correctness property, which holds for these two runs. Due to a potential race condition between t_{13} and t_{25}, the assertion may fail on a run such as $t_{10}t_{11}\{t_{21} \cdots t_{24}\}t_{12}t_{25}t_{13}\{t_{16} \cdots t_{19}\}$ with $x = 2, G = 0, H = 0$.

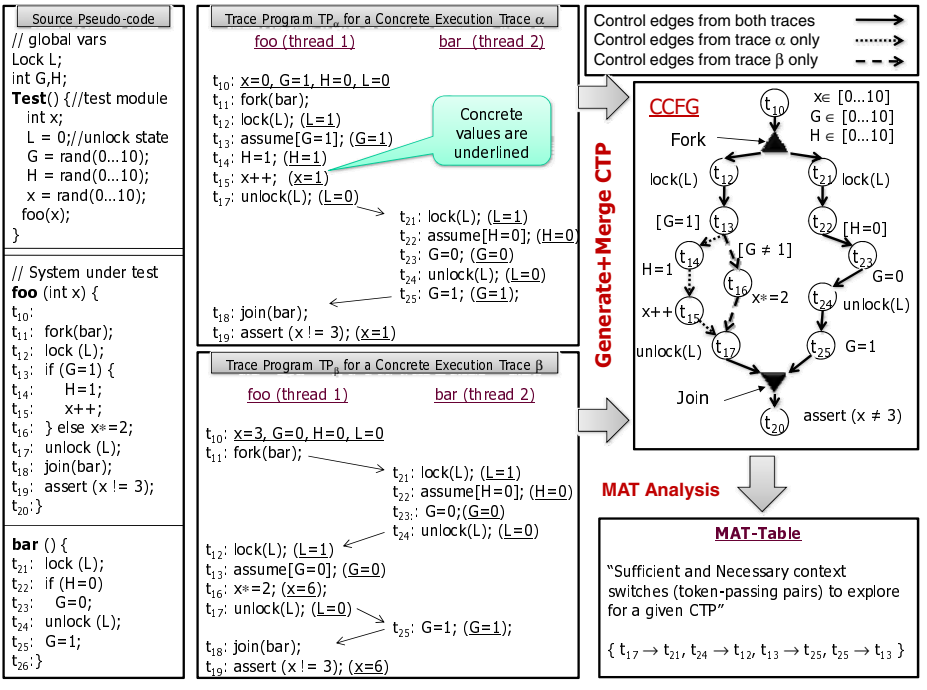


Fig. 2. A run of the tool

Generate CTP. From a trace, we obtain a concurrent trace program (CTP) by relaxing the order (of events) induced by non-deterministic scheduling, and maintaining only those that are induced by thread program order and fork/join semantics [7]. For the example, fork/join induces following partial-order event pairs: (t_{11}, t_{12}), (t_{11}, t_{21}), (t_{25}, t_{18}), and (t_{17}, t_{18}). We also allow data inputs to take range values as defined in the test harness. As shown, x, G and H take arbitrary values within the range $[0 \cdots 10]$.

We represent CTP_α (CTP_β), as a concurrent control flow graph $CCFG_\alpha$ ($CCFG_\beta$) (not shown separately). The control edges in a CCFG represent the partially-ordered events of the CTP. These two CCFGs can be "stitched" together by merging the respective control edges. This combined CCFG implicitly represents a "merged" CTP. The solid arrows denote the control edges common in both CTPs, whereas the dotted/dashed arrows denote the exclusive control edges from CTP_α/CTP_β, respectively.

Symbolic Analysis. All linearizations of a CTP may not correspond to actual executions of the program. For example, a linearization $\cdots t_{25} t_{13} t_{16} \cdots$ does not correspond to any executable trace (as the branch $[G \neq 1]$ will not hold). We define "feasible linearizations" of a CTP to be those that correspond to actual program executions [7]. We generate verification conditions (using the following encoding) to search within the feasible linearizations of a CTP; if an error is found, it is guaranteed to be real.

SMT-based Encoding (`mat-enc`). We use a token-passing modeling approach [9] to generate a quantifier-free first-order logic formula (ϕ). First, we create independent (uncoupled) models for each individual thread in the CTP. On such thread-models, we apply thread-local transformation and simplification [9] to reduce the thread modeling constraints (ϕ_{TM}). Second, we add token passing constraints ϕ_{TPM} between only those context-switching events as identified by MAT analysis (described next). Optionally, we add thread-specific fine grained context-bounding constraints ϕ_{CB} for more scalability, though at the cost of completeness. The formula ϕ represents the following conjunction, where ϕ_{PRP} denote the formula corresponding to the correctness property.

$$\phi = \phi_{TM} \wedge \phi_{TPM} \wedge \neg\phi_{PRP}(\ \wedge \ \phi_{CB} \ )$$

MAT Analysis. On a CTP, we use a partial-order reduction technique based on *Mutually Atomic Transactions* (MAT) [8] to identify an optimal and adequate set of context switches (or token-passing pairs) to cover the entire interleaving space of the CTP. The basic idea is as follows: a MAT is a pair of transactions (i.e., a sequence of transitions) corresponding to two threads, such that only the last transitions in the pair of transactions have conflicting shared variable accesses. An interesting observation is that there are only two different program behaviors possible by interleaving the various transitions in a MAT. As shown for the example, MAT analysis produces a necessary set of only 4 context switches, whose combination guarantees a complete thread interleaving coverage for the CTP. Such a reduced set of context switches not only reduces the interleaving search space but also the size of the formula ϕ. Due to this improvement, the tool can search a potentially a larger CTP (i.e., a larger coverage) for possible violations.

3 Evaluation

We applied CONTESSA to several case studies. In one case study [8], our goal was to check assertions that specify the functional correctness of multi-threaded programs. We used random test vectors to generate CTPs of trace depths of 400. Note, though each test vector may produce a distinct trace (CCFG), typically it differs from the rest only in a few control edges. By stitching these CCFGs, as mentioned earlier, we obtain a compact CTP. In these CTPs, the number of threads was 2 to 3, the number of shared accesses was between 4 and 200 per thread. The total number of possible context-switches was between 50-800K. After MAT analysis, the number of context-switches was reduced to 14-2500. This reduction directly translated in the size reduction of verification conditions from the range 32K-48M to 26K-3.3M. On a few of these CTPs, we found assertion violations in less than a minute.

In another case study, we applied our tool to obtain CTPs from execution traces generated by Java PathFinder [11]. The test programs include publicly available multi-threaded Java benchmarks such as *hdec*, *Daisy*, and *Tsp*. The trace lengths range from 200 to 45K, and the number of threads ranges from 3 to 21. In these generated CTPs, the number of lock/unlock events ranges from 4 to 1K, and the number of wait/notify events ranges from 0 to 41. Our symbolic analysis algorithm were able to find data races in a few of these CTPs within one minute, producing corresponding witness traces. Thus,

our tool can effectively be applied as a plug-in module in prevalent testing infrastructures to improve the test coverage of concurrent programs.

Overall, we believe that the tool is a promising compromise between scalability of testing and coverage of symbolic static analysis.

References

1. Godefroid, P.: Software Model Checking: The Verisoft approach. In: FMSD (2005)
2. Musuvathi, M., Quadeer, S.: CHESS: Systematic stress testing of concurrent software. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 15–16. Springer, Heidelberg (2007)
3. Yang, Y., Chen, X., Gopalakrishnan, G.: Inspect: A Runtime Model Checker for Multithreaded C Programs. Technical Report UUCS-08-004, University of Utah (2008)
4. Edelstein, O., Farchi, E., Goldin, E., Nir, Y., Ratsaby, G., Ur, S.: Framework for Testing Multi-threaded Java Programs. In: Concurrency and Computation: Practice and Experience (2003)
5. Sen, K.: Race directed random testing of concurrent programs. In: PLDI (2008)
6. Farzan, A., Madhusudan, P.: Causal Atomicity. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 315–328. Springer, Heidelberg (2006)
7. Wang, C., Chaudhuri, S., Gupta, A., Yang, Y.: Symbolic pruning of concurrent program executions. In: ESEC-FSE (2009)
8. Ganai, M.K., Kundu, S.: Reduction of Verification Conditions for Concurrent System using Mutually Atomic Transactions. In: Proc. of SPIN Workshop (2009)
9. Ganai, M.K., Gupta, A.: Efficient modeling of concurrent systems in bmc. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 114–133. Springer, Heidelberg (2008)
10. SRI. Yices: An SMT solver, http://fm.csl.sri.com/yices
11. JPF, http://babelfish.arc.nasa.gov/trac/jpf

Simulation Subsumption in Ramsey-Based Büchi Automata Universality and Inclusion Testing*

Parosh Aziz Abdulla[1], Yu-Fang Chen[2], Lorenzo Clemente[3], Lukáš Holík[4], Chih-Duo Hong[2], Richard Mayr[3], and Tomáš Vojnar[4]

[1] Uppsala University
[2] Academia Sinica
[3] University of Edinburgh
[4] Brno University of Technology

Abstract. There are two main classes of methods for checking universality and language inclusion of Büchi-automata: Rank-based methods and Ramsey-based methods. While rank-based methods have a better worst-case complexity, Ramsey-based methods have been shown to be quite competitive in practice [10,9]. It was shown in [10] (for universality checking) that a simple subsumption technique, which avoids exploration of certain cases, greatly improves the performance of the Ramsey-based method. Here, we present a much more general subsumption technique for the Ramsey-based method, which is based on using simulation preorder on the states of the Büchi-automata. This technique applies to both universality and inclusion checking, yielding a substantial performance gain over the previous simple subsumption approach of [10].

1 Introduction

Universality and language-inclusion checking are important problems in the theory of automata with significant applications, e.g., in model-checking. More precisely, the problem of checking whether an implementation meets a specification can be formulated as a language inclusion problem. The behavior of the implementation is represented by an automaton $\mathcal{A}$, the specification is given by an automaton $\mathcal{B}$, and one checks whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. As one is generally interested in non-halting computations, automata are used as acceptors of languages over *infinite words*. In this paper, we concentrate on *Büchi automata*, where accepting runs are those containing some accepting state infinitely often.

A naïve inclusion-checking algorithm involves complementation: One has that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{B})} = \emptyset$. However, the complementary automaton $\overline{\mathcal{B}}$ is,

* This work was supported in part by the Royal Society grant JP080268, the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the internal BUT FIT grant FIT-10-1, the UPMARC project, the CONNECT project, National Science Council of Taiwan project no. 99-2218-E-001-002-MY3, and the ESF project Games for Design and Verification.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 132–147, 2010.

in the worst case, exponentially bigger than the original automaton $\mathcal{B}$. Hence, direct complementation should be avoided.

Among methods that keep the complementation step implicit, *Rank-based* and *Ramsey-based* methods have recently gained interest. The former uses a rank-based analysis of rejecting runs [14], leading to a simplified complementation procedure. The latter is based on Büchi's original combinatorial Ramsey-based argument for showing closure of ω-regular languages under complementation [4]. Notice that a high worst-case complexity is unavoidable, since both universality and language-inclusion testing are PSPACE-complete problems.

However, in practice, subsumption techniques can often greatly speed up universality/inclusion checking by avoiding the exploration of certain cases that are subsumed by other cases. Recently, [5] described a simple set-inclusion-based subsumption technique that speeds up the rank-based method for both universality and language inclusion checking. Using this technique, [5] is capable of handling automata several orders of magnitude larger than previously possible. Similarly, [10] improved the Ramsey-based method (but only for universality checking) by a simple subsumption technique that compares finite labeled graphs (using set-inclusion on the set of arcs, plus an order on the labels; see the last paragraph in Section 3).

Our contribution. We improve the Ramsey-based approach in a twofold way. First, we show how to employ simulation preorder to generalize the simple subsumption technique of [10] for Ramsey-based universality checking. Second, we introduce a simulation-based subsumption relation for Ramsey-based language inclusion checking, thus extending the theory of subsumption to the realm of Ramsey-based inclusion checking. Note that the proposed use of simulations is significantly different from just using simulations to reduce Büchi automata (quotienting), followed by an application of the original approach. The reason is that, when reducing automata, one has to use simulation equivalences which tend to be much smaller (and hence less helpful) than simulation preorders, which are used in our method. Therefore, our approach takes full advantage of the asymmetry of simulation preorder, making it more general than just quotienting beforehand the automaton w.r.t. the induced equivalence.

Experimental results show that our algorithm based on simulation subsumption significantly and consistently outperforms the algorithm based on the original subsumption of [10]. We perform the evaluation on Büchi automata models of several mutual exclusion algorithms (the largest having several thousands of states and tens of thousands of transitions), random Büchi automata generated from LTL formulae, and Büchi automata generated from the random model of [18]. In many cases, the difference between the two approaches is very significant. For example, our approach finishes an experiment on the Bakery algorithm in minutes, while the original approach cannot handle it in hours. In the largest examples generated from LTL formulae, our approach is on average 20 times faster than the original one when testing universality and more than 1900 times faster when testing language inclusion. All relevant information is provided online [21], enabling interested readers to reproduce our experiments.

2 Preliminaries

A *Büchi Automaton (BA)* $\mathcal{A}$ is a tuple $(\Sigma, Q, I, F, \delta)$ where Σ is a finite alphabet, Q is a finite set of states, $I \subseteq Q$ is a non-empty set of *initial* states, $F \subseteq Q$ is a set of *accepting* states, and $\delta \subseteq Q \times \Sigma \times Q$ is the transition relation. For convenience, we write $p \xrightarrow{a} q$ instead of $(p, a, q) \in \delta$.

A *run* of $\mathcal{A}$ on a word $w = a_1 a_2 \ldots \in \Sigma^\omega$ *starting* in a state $q_0 \in Q$ is an infinite sequence $q_0 q_1 q_2 \ldots$ such that $q_{j-1} \xrightarrow{a_j} q_j$ for all $j > 0$. The run is *accepting* iff $q_i \in F$ for infinitely many i. The *language of* $\mathcal{A}$ is the set $\mathcal{L}(\mathcal{A}) = \{w \in \Sigma^\omega \mid \mathcal{A} \text{ has an accepting run on } w \text{ starting from some } q_0 \in I\}$.

A *path* in $\mathcal{A}$ on a finite word $w = a_1 \ldots a_n \in \Sigma^+$ is a finite sequence $q_0 q_1 \ldots q_n$ where $q_{j-1} \xrightarrow{a_j} q_j$ for all $0 < j \leq n$. The path is *accepting* iff $q_i \in F$ for some $0 \leq i \leq n$. We define the following predicates for $p, q \in Q$: (1) $p \overset{w}{\underset{F}{\Longrightarrow}} q$ iff there is an accepting path on w from p to q. (2) $p \overset{w}{\Longrightarrow} q$ iff there is a path (not necessarily accepting) on w from p to q. (3) $p \overset{w}{\not\Longrightarrow} q$ iff there is no path on w from p to q.

Define $E = Q \times \{0, 1, -1\} \times Q$ and let $G_\mathcal{A}$ be the largest subset of 2^E whose elements contain exactly one member of $\{\langle p, 0, q\rangle, \langle p, 1, q\rangle, \langle p, -1, q\rangle\}$ for every $p, q \in Q$. Each element in $G_\mathcal{A}$ is a $\{0, 1, -1\}$-arc-labeled graph on Q.

For each pair of states $p, q \in Q$, we define the following three sets of languages: (1) $\mathcal{L}(p, 1, q) = \{w \in \Sigma^+ \mid p \overset{w}{\underset{F}{\Longrightarrow}} q\}$, (2) $\mathcal{L}(p, 0, q) = \{w \in \Sigma^+ \mid p \overset{w}{\Longrightarrow} q \wedge \neg\left(p \overset{w}{\underset{F}{\Longrightarrow}} q\right)\}$, (3) $\mathcal{L}(p, -1, q) = \{w \in \Sigma^+ \mid p \overset{w}{\not\Longrightarrow} q\}$. As in [10], the language of a graph $g \in G_\mathcal{A}$ is defined as the intersection of the languages of arcs in g, i.e., $\mathcal{L}(g) = \bigcap_{\langle p,a,q\rangle \in g} \mathcal{L}(p, a, q)$. For each word $w \in \Sigma^+$ and each pair $p, q \in Q$, there exists exactly one arc $\langle p, a, q\rangle$ such that $w \in \mathcal{L}(p, a, q)$. Therefore, the languages of the graphs in $G_\mathcal{A}$ form a partition of Σ^+, since they are the intersections of the languages of the arcs. Let Y_{gh} be the ω-regular language $\mathcal{L}(g) \cdot \mathcal{L}(h)^\omega$.

Lemma 1. *(1)* $\Sigma^\omega = \bigcup_{g,h \in G_\mathcal{A}} Y_{gh}$. *(2) For* $g, h \in G_\mathcal{A}$ *s.t.* $\mathcal{L}(g), \mathcal{L}(h) \neq \emptyset$, *either* $Y_{gh} \cap \mathcal{L}(\mathcal{A}) = \emptyset$ *or* $Y_{gh} \subseteq \mathcal{L}(\mathcal{A})$. *(3)* $\overline{\mathcal{L}(\mathcal{A})} = \bigcup_{g,h \in G_\mathcal{A} \wedge Y_{gh} \cap \mathcal{L}(\mathcal{A}) = \emptyset} Y_{gh}$.

In fact, Lemma 1 is a relaxed version of the lemma proved by a Ramsey-based argument described in [16,9,10]. A proof can be found in [1].

3 Ramsey-Based Universality Testing

Based on Lemma 1, one can construct an algorithm for checking universality of BA [9]. This type of algorithm is said to be Ramsey-based, since the proof of Lemma 1 relies on the infinite Ramsey theorem. Lemma 1 implies that $\mathcal{L}(\mathcal{A})$ is universal iff $\forall g, h \in G_\mathcal{A} : \; Y_{gh} \subseteq \mathcal{L}(\mathcal{A})$. Since $\mathcal{L}(g) = \emptyset$ or $\mathcal{L}(h) = \emptyset$ implies $Y_{gh} \subseteq \mathcal{L}(\mathcal{A})$, it suffices to build and check graphs with nonempty languages in $G_\mathcal{A}$ when testing universality.

As proposed in [9,10,13], the set $G^f_{\mathcal{A}} = \{g \in G_{\mathcal{A}} \mid \mathcal{L}(g) \neq \emptyset\}$ can be generated iteratively as follows. First, given $g, h \in G_{\mathcal{A}}$, their composition $g;h$ is defined as

$$\begin{aligned}
&\{\langle p, -1, q\rangle \mid \forall t \in Q : (\langle p, a, t\rangle \in g \wedge \langle t, b, q\rangle \in h) \rightarrow (a = -1 \vee b = -1)\} \cup \\
&\{\langle p, 0, q\rangle \mid \exists r \in Q : \langle p, 0, r\rangle \in g \wedge \langle r, 0, q\rangle \in h \wedge \\
&\qquad \wedge\ \forall t \in Q : (\langle p, a, t\rangle \in g \wedge \langle t, b, q\rangle \in h) \rightarrow (a \neq 1 \wedge b \neq 1)\} \cup \\
&\{\langle p, 1, q\rangle \mid \exists r \in Q : \langle p, a, r\rangle \in g \wedge \langle r, b, q\rangle \in h \wedge \neg(a \neq 1 \wedge b \neq 1)\}.
\end{aligned}$$

For all $a \in \Sigma$, define the single-character graph $g_a = \{\langle p, -1, q\rangle \mid q \notin \delta(p, a)\} \cup \{\langle p, 0, q\rangle \mid p \in (Q \setminus F) \wedge q \in (\delta(p, a) \setminus F)\} \cup \{\langle p, 1, q\rangle \mid q \in \delta(p, a) \wedge \{p, q\} \cap F \neq \emptyset\}$. Let $G^1_{\mathcal{A}} = \{g_a \mid a \in \Sigma\}$. As shown in [8] (Lemma 3.1.1), one can obtain $G^f_{\mathcal{A}}$ by repeatedly composing graphs in $G^1_{\mathcal{A}}$ until a fixpoint is reached:

Lemma 2. *A graph g is in $G^f_{\mathcal{A}}$ iff $\exists g_1, \ldots, g_n \in G^1_{\mathcal{A}} :\ g = g_1; \ldots; g_n$.*

It remains to sketch how to check that no pair $\langle g, h\rangle$ of graphs $g, h \in G^f_{\mathcal{A}}$ is a counterexample to universality, which, by Point 3 of Lemma 1, reduces to testing $Y_{gh} \cap \mathcal{L}(\mathcal{A}) \neq \emptyset$. The so called *lasso-finding test*, proposed in [10], can be used for this purpose. The lasso-finding test of a pair of graphs $\langle g, h\rangle$ checks the existence of a *lasso*, i.e., a path in g from some state $p \in I$ to some state $q \in F$ and a path in h from q to itself. Equivalently, we consider a pair of graphs $\langle g, h\rangle$ to pass the *lasso-finding test* (denoted by $LFT(g, h)$) iff there is an arc $\langle p, a_0, q_0\rangle$ in g and an infinite sequence of arcs $\langle q_0, a_1, q_1\rangle, \langle q_1, a_2, q_2\rangle, \ldots$ in h s.t. $p \in I$, $a_i \in \{0, 1\}$ for all $i \geq 0$, and $a_j = 1$ for infinitely many $j \in \mathbb{N}$. The following lemma was proved in [10] (we provide a considerably simplified proof in [1]).

Lemma 3. *$\mathcal{L}(\mathcal{A})$ is universal iff $LFT(g, h)$ for all $g, h \in G^f_{\mathcal{A}}$.*

To be more specific, the procedure for the lasso-finding test works as follows. It (1) finds all 1-SCCs (strongly connected components that contain only $\{0, 1\}$-labeled arcs and at least one of the arcs is 1-labeled) in h, (2) records the set of states T_h from which there is an $\{0, 1\}$-labeled path to some state in some 1-SCC, (3) records the set of states H_g such that for all $q \in H_g$, there exists an arc $\langle p, a, q\rangle \in g$ for some $p \in I$ and $a \geq 0$, and then (4) checks if $H_g \cap T_h \neq \emptyset$. We have $LFT(g, h)$ iff $H_g \cap T_h \neq \emptyset$. This procedure is *polynomial* in the number of $\{0, 1\}$-labeled arcs in g and h.

Finally, Algorithm 1 gives a naïve universality test obtained by combining the above principles for generating $G^f_{\mathcal{A}}$ and using LFT. A more efficient version of the algorithm is given in [10], using the following idea. For $f, g, h \in G_{\mathcal{A}}$, we say that $g \sqsubseteq h$ iff for each arc $\langle p, a, q\rangle \in g$, there is an arc $\langle p, a', q\rangle \in h$ such that $a \leq a'$. If $g \sqsubseteq h$, we have that (1) $LFT(f, g) \implies LFT(f, h)$ and (2) $LFT(g, f) \implies LFT(h, f)$ for all $f \in G_{\mathcal{A}}$. Since the algorithm searches for counterexamples to universality, the tests on h are subsumed by the tests on g, and thus h can be discarded. We refer to this method, which is based on the relation $\sqsubseteq$, as *subsumption*, in contrast to our more general *simulation subsumption* which is described in the next section.

Algorithm 1. *Ramsey-based Universality Checking*

Input: A BA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, the set of all single-character graphs $G^1_{\mathcal{A}}$
Output: TRUE if $\mathcal{A}$ is universal. Otherwise, FALSE.

```
1 Next := G^1_A; Processed := ∅;
2 while Next ≠ ∅ do
3     Pick and remove a graph g from Next;
4     foreach h ∈ Processed do
5         if ¬LFT(g,h) ∨ ¬LFT(h,g) ∨ ¬LFT(g,g) then return FALSE;
6     Add g to Processed;
7     foreach h ∈ G^1_A do if g;h ∉ Processed then Add g;h to Next;
8 return TRUE;
```

4 Improving Universality Testing via Simulation

In this section, we describe our technique to use simulation-based subsumption in order to accelerate the Ramsey-based universality test [10] for Büchi automata.

A *simulation* on a BA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$ is a relation $R \subseteq Q \times Q$ such that pRr only if (1) $p \in F \implies r \in F$, and (2) for every transition $p \xrightarrow{a} p'$, there is a transition $r \xrightarrow{a} r'$ such that $p'Rr'$. It can be shown that there exists a unique maximal simulation, which is a preorder (called *simulation preorder* and denoted by $\preceq_{\mathcal{A}}$, or just $\preceq$ when $\mathcal{A}$ is clear from the context), computable in time $\mathcal{O}(|\Sigma||Q||\delta|)$ [11,12]. The relation $\simeq = \preceq \cap \succeq$ is called *simulation equivalence*.

If $\mathcal{A}$ is interpreted as an automaton over finite words, $\preceq$ implies language containment, and quotienting w.r.t. $\simeq$ preserves the regular language. If $\mathcal{A}$ is interpreted as a BA, then the particular type of simulation defined above is called *direct simulation*. It implies ω-language containment, and (unlike for fair simulation [7]) quotienting w.r.t. $\simeq$ preserves the ω-regular language of $\mathcal{A}$.

Our method for accelerating the Ramsey-based universality test [10] of $\mathcal{A}$ is based on two optimizations which we describe below together with some intuition underlying their correctness. We formally prove the correctness of these optimizations in Lemmas 4–8, presented afterwards.

Optimization 1. The first optimization is based on the observation that the subsumption relation $\sqsubseteq$ from [10] can be weakened by exploiting simulation preorder. We call our weaker notion the *simulation-subsumption-based relation*, written $\sqsubseteq^{\forall\exists}$. The idea is as follows: While in $\sqsubseteq$ one requires that arcs of the form $\langle p, a, q\rangle$ can only be subsumed by arcs of the form $\langle p, a', q'\rangle$ with $a \leq a' \wedge q' = q$, we generalize this notion by replacing the last equality with simulation. This gives rise to the definition of $\sqsubseteq^{\forall\exists}$ below.

Definition 1. *For any $g, h \in G_{\mathcal{A}}$, we say that $g \sqsubseteq^{\forall\exists} h$ iff for every arc $\langle p, a, q\rangle \in g$, there exists an arc $\langle p, a', q'\rangle \in h$ such that $a \leq a'$ and $q \preceq q'$.*

Optimization 2. The second optimization builds on the fact that even the structure of the particular graphs in $G_{\mathcal{A}}$ can be simplified via simulation-subsumption,

allowing us to replace some $\{0,1\}$-labeled arcs by negative arcs. Since the complexity of the lasso-finding test, subsumption-checking, and graph-composition is polynomial in the number of $\{0,1\}$-arcs, having smaller graphs reduces the cost of these operations.

For the purpose of reducing graphs, we define a (possibly non-deterministic) operation *Min* that maps each graph $f \in G_{\mathcal{A}}$ to a graph $Min(f) \in G_{\mathcal{A}}$ such that $Min(f) \leqslant^* f$. Here, $g \leqslant^* h$ means that g is either equal to h, or it is a reduced version of h that can be derived from h by weakening some of the arcs that are subsumed (simulation-smaller) by other arcs present both in g and h. We define $\leqslant^*$ below.

Definition 2. *For any graphs $g, h \in G_{\mathcal{A}}$, we write $g \leqslant h$ iff there exist arcs $\langle p, a, q\rangle \in h$ and $\langle p, a', q'\rangle \in g \cap h$ s.t. $a \leq a'$, $q \preceq q'$, and $g = (h \setminus \{\langle p, a, q\rangle\}) \cup \{\langle p, a'', q\rangle\}$ where $a'' \leq a$. The relation $\leqslant^*$ is the transitive closure of $\leqslant$.*

We write $G^m_{\mathcal{A}} = \{g \in G_{\mathcal{A}} \mid \exists h \in G^f_{\mathcal{A}} : \ g \leqslant^* h\}$ to denote the set of reduced versions of graphs with nonempty languages. In practice, *Min* can be implemented such that it returns a graph which is as $\leqslant^*$-small as possible (meaning that as many arcs as possible will be restricted down to -1).

Correctness of the Optimizations. We now prove the correctness of our optimizations in a formal way. The correctness of the second optimization follows directly from 1) the observation that $\leqslant$ implies $\sqsubseteq^{\forall\exists}$-equivalence (Lemma 4), and 2) the fact that $G^m_{\mathcal{A}}$ is closed under composition (Lemma 8).

Let $\simeq^{\forall\exists} = \sqsubseteq^{\forall\exists} \cap (\sqsubseteq^{\forall\exists})^{-1}$. The lemma below follows by transitivity.

Lemma 4. *For any $g, h \in G_{\mathcal{A}}$, if $g \leqslant^* h$ then $g \simeq^{\forall\exists} h$.*

In particular, minimized graphs are $\sqsubseteq^{\forall\exists}$-equivalent to their original version. Notice that the relation $\leqslant^*$ does not preserve the language of graphs (and often for $g \leqslant^* h$, $\mathcal{L}(g) = \emptyset$ when $\mathcal{L}(h) \neq \emptyset$).

The correctness of the first optimization follows from the following observations. First, the lasso-finding test is $\sqsubseteq^{\forall\exists}$-monotonic, i.e., if $\sqsubseteq^{\forall\exists}$-smaller (pairs of) graphs pass the test, then so do $\sqsubseteq^{\forall\exists}$-bigger (pairs of) graphs (Lemma 7). In particular, for graphs $f, g, h \in G^f_{\mathcal{A}}$ such that $g \sqsubseteq^{\forall\exists} h$, $LFT(g, f) \implies LFT(h, f)$ and $LFT(f, g) \implies LFT(f, h)$. Therefore, we can ignore all lasso-finding tests related to the bigger h. Second, graph-composition is also $\sqsubseteq^{\forall\exists}$-monotonic (Lemma 6): Composing $\sqsubseteq^{\forall\exists}$-smaller graphs always yields $\sqsubseteq^{\forall\exists}$-smaller graphs. Thus, we can also ignore all lasso-finding tests related to any extension $h; f$ of h, for some $f \in G^f_{\mathcal{A}}$.

We begin by proving an auxiliary lemma—used to prove Lemma 6—which says that minimized graphs are in some sense complete w.r.t. simulation-bigger states.

Lemma 5. *Let g be a graph in $G^m_{\mathcal{A}}$. We have that $\langle p, a, q\rangle \in g \ \wedge \ p \preceq p'$ implies $\exists\langle p', a', q'\rangle \in g : \ a \leq a' \ \wedge \ q \preceq q'$.*

Proof. If $a = -1$, the lemma trivially holds (e.g., by taking $q' = q$). Assume therefore $a \in \{0,1\}$. From $g \in G^m_{\mathcal{A}}$, there is some $g' \in G^f_{\mathcal{A}}$ such that $g \leqslant^* g'$. Since $\mathcal{L}(g') \neq \emptyset$ and $a \in \{0,1\}$, there is some word $w \in \mathcal{L}(g')$ such that $p \xRightarrow{w} q$. Since $p \preceq p'$, there is some q'' such that $p' \xRightarrow{w} q''$, $q \preceq q''$, and if $p \xRightarrow[F]{w} q$, then $p' \xRightarrow[F]{w} q''$. Since $w \in \mathcal{L}(g')$, $\langle p', a'', q'' \rangle \in g'$ for $a \leq a''$. From Lemma 4, we get that there is an arc $\langle p', a', q' \rangle \in g$ such that $a \leq a'' \leq a'$ and $q \preceq q'' \preceq q'$. □

The lemma below states that composing minimized graphs is a $\sqsubseteq^{\forall\exists}$-monotonic operation. We actually prove a slightly stronger property, since we do not require that all graphs are minimal.

Lemma 6. *Let $f, g, f' \in G_{\mathcal{A}}$ and $g' \in G^m_{\mathcal{A}}$ be graphs s.t. $f \sqsubseteq^{\forall\exists} f'$ and $g \sqsubseteq^{\forall\exists} g'$. Then $f;g \sqsubseteq^{\forall\exists} f';g'$.*

Proof. We consider an arc $\langle p, c, r \rangle$ in $f;g$ and show that $f';g'$ must contain a larger arc w.r.t. $\sqsubseteq^{\forall\exists}$. The case $c = -1$ is trivial. For $c \in \{0,1\}$, there must be arcs $\langle p, a, q \rangle \in f$ and $\langle q, b, r \rangle \in g$ where $a, b \in \{0,1\}$ and $c = \max(\{a, b\})$. Since $f \sqsubseteq^{\forall\exists} f'$, there is an arc $\langle p, a', q' \rangle \in f'$ with $a \leq a'$ and $q \preceq q'$. Since $g \sqsubseteq^{\forall\exists} g'$, there is an arc $\langle q, b', r' \rangle \in g'$ with $b \leq b'$ and $r \preceq r'$. Since $g' \in G^m_{\mathcal{A}}$, Lemma 5 implies that there is an arc $\langle q', b'', r'' \rangle \in g'$ s.t. $b \leq b' \leq b''$ and $r \preceq r' \preceq r''$. Thus $\langle p, c'', r'' \rangle \in f';g'$ where $c = \max(\{a, b\}) \leq \max(\{a', b''\}) \leq c''$ and $r \preceq r''$. □

Below, we prove a lemma allowing us to replace lasso-finding tests on graphs by lasso-finding tests on (minimized versions of) smaller graphs. For the sake of generality, we prove a somewhat stronger property.

Lemma 7. *Let e, f, g, h be graphs in $G_{\mathcal{A}}$ such that $\{f, h\} \cap G^m_{\mathcal{A}} \neq \emptyset$, $e \sqsubseteq^{\forall\exists} g$, and $f \sqsubseteq^{\forall\exists} h$. Then $LFT(e,f) \implies LFT(g,h)$.*

Proof. If $LFT(e,f)$, there exist an arc $\langle p, a_0, q_0 \rangle \in e$ and an infinite sequence of arcs $\langle q_0, a_1, q_1 \rangle$, $\langle q_1, a_2, q_2 \rangle$, ... in f s.t. $p \in I$, $a_i \in \{0,1\}$ for all i, and $a_j = 1$ for infinitely many j. By the premise $e \sqsubseteq^{\forall\exists} g$, there is $\langle p, a'_0, q'_0 \rangle \in g$ s.t. $a_0 \leq a'_0$ and $q_0 \preceq q'_0$ (Property 1). We now show how to construct an infinite sequence $q'_0 a'_1 q'_1 a'_2 q'_2 \cdots$ that satisfies the following (Property 2): $\langle q'_n, a'_{n+1}, q'_{n+1} \rangle \in h$, $a_{n+1} \leq a'_{n+1}$, and $q_n \preceq q'_n$ for all $n \geq 0$. We do this by proving that every finite sequence $q'_0 a'_1 q'_1 \ldots q'_{k-1} a'_k q'_k$ satisfying Property 2 can be extended by one step to length $k+1$ while preserving Property 2. Moreover, such a sequence can be started (case $k = 0$) since for $k = 0$, Property 1 implies Property 2 as q'_1 is not in the sequence then. For the extension, we distinguish two (non-exclusive) cases:

1. $f \in G^m_{\mathcal{A}}$. Since $\langle q_k, a_{k+1}, q_{k+1} \rangle \in f$ and $q_k \preceq q'_k$ (by Property 2), Lemma 5 implies that there exists an arc $\langle q'_k, a, q \rangle \in f$ such that $a_{k+1} \leq a$ and $q_{k+1} \preceq q$. Since $f \sqsubseteq^{\forall\exists} h$, there must be some arc $\langle q'_k, a'_{k+1}, q'_{k+1} \rangle \in h$ such that $a_{k+1} \leq a \leq a'_{k+1}$ and $q_{k+1} \preceq q \preceq q'_{k+1}$.
2. $h \in G^m_{\mathcal{A}}$. Since $\langle q_k, a_{k+1}, q_{k+1} \rangle \in f$ and $f \sqsubseteq^{\forall\exists} h$, there is some arc $\langle q_k, a, q \rangle \in h$ s.t. $a_{k+1} \leq a$ and $q_{k+1} \preceq q$. Since $q_k \preceq q'_k$ (by Property 2) and $\langle q_k, a, q \rangle \in h$, Lemma 5 implies that there is an arc $\langle q'_k, a'_{k+1} q'_{k+1} \rangle \in h$ such that $a_{k+1} \leq a \leq a'_{k+1}$ and $q_{k+1} \preceq q \preceq q'_{k+1}$.

To conclude, there exist an arc $\langle p, a'_0, q'_0\rangle \in g$ and an infinite sequence of arcs $\langle q'_0, a'_1, q'_1\rangle$, $\langle q'_1, a'_2, q'_2\rangle$, ... in h such that $p \in I$ and $a'_i \in \{0,1\}$ for all i and $a'_j = 1$ for infinitely many j. Hence, $LFT(g,h)$ holds. □

Finally, we show that the set $G^m_{\mathcal{A}}$ is closed under composition.

Lemma 8. *$G^m_{\mathcal{A}}$ is closed under composition. That is, $\forall e, f \in G^m_{\mathcal{A}} : e; f \in G^m_{\mathcal{A}}$.*

Proof. As $e, f \in G^m_{\mathcal{A}}$, there are $g, h \in G^f_{\mathcal{A}}$ with $e \leqslant^* g$ and $f \leqslant^* h$. We will show that $e; f \leqslant^* g; h$ (notice that this does not directly follow from Lemma 6, since $\sqsubseteq^{\forall\exists}$ does not imply $\leqslant^*$). Since by Lemma 2, $g; h \in G^f_{\mathcal{A}}$, this will give $e; f \in G^m_{\mathcal{A}}$. By the definition of $\leqslant^*$, there are $g_0, h_0, g_1, h_1, \ldots, g_n, h_n \in G^m_{\mathcal{A}}$ s.t. $g_0 = g, h_0 = h, g_n = e, h_n = f$, and for each $i : 1 \leq i \leq n$, $g_i \leqslant g_{i-1}$ and $h_i \leqslant h_{i-1}$. We will show that for any $i : 1 \leq i \leq n$, $g_i; h_i \leqslant^* g_{i-1}; h_{i-1}$ which implies that $e; f \leqslant^* g; h$.

Since $g_i \leqslant g_{i-1}$, for every arc $\langle p, a, q\rangle \in g_i$, $\langle p, a', q\rangle \in g_{i-1}$ with $a \leq a'$. Since $h_i \leqslant h_{i-1}$, for every arc $\langle q, b, r\rangle \in h_i$, $\langle q, b', r\rangle \in h_{i-1}$ with $b \leq b'$. Therefore, by the definition of composition, for each $\langle p, c, r\rangle \in g_i; h_i$, we have $\langle p, c', r\rangle \in g_{i-1}; h_{i-1}$ with $c \leq c'$. To prove that $g_i; h_i \leqslant^* g_{i-1}; h_{i-1}$, it remains to show that there is also $\langle p, \bar{c}, \bar{r}\rangle \in g_i; h_i \cap g_{i-1}; h_{i-1}$ with $c' \leq \bar{c}$ and $r \preceq \bar{r}$. The case when $c = c'$ is trivial. If $c < c'$, then $0 \leq c'$ and thus there are $\langle p, a, q\rangle \in g_{i-1}$ and $\langle q, b, r\rangle \in h_{i-1}$ s.t. $c' = \max(\{a, b\})$. Since $g_i \leqslant g_{i-1}$, there is $\langle p, \bar{a}, \bar{q}\rangle \in g_i \cap g_{i-1}$ with $a \leq \bar{a}$ and $q \preceq \bar{q}$. By Lemma 5 and as $h_i \in G^m_{\mathcal{A}}$, there is also $\langle \bar{q}, b', r'\rangle \in h_i$ with $b \leq b'$ and $r \preceq r'$. Since $h_i \leqslant h_{i-1}$, there is $\langle \bar{q}, \bar{b}, \bar{r}\rangle \in h_i \cap h_{i-1}$ where $b' \leq \bar{b}$ and $r' \preceq \bar{r}$. Together with $\langle p, \bar{a}, \bar{q}\rangle \in g_i \cap g_{i-1}$, this implies that there is $\langle p, \bar{c}, \bar{r}\rangle \in g_i; h_i \cap g_{i-1}; h_{i-1}$ with $\max(\{\bar{a}, \bar{b}\}) \leq \bar{c}$ and $r' \preceq \bar{r}$. Since $c' = \max(\{a, b\}) \leq \max(\{\bar{a}, \bar{b}\}) \leq \bar{c}$ and $r \preceq r' \preceq \bar{r}$, $\langle p, \bar{c}, \bar{r}\rangle$ is the wanted arc. □

The Algorithm. Algorithm 2 gives a complete description of our approach to universality testing of BA. In this algorithm, Lines 4, 5, 14, and 15 implement Optimization 1; Lines 1 and 13 implement Optimization 2. Overall, the algorithm works such that for each graph in $G^f_{\mathcal{A}}$, a minimization of some $\sqsubseteq^{\forall\exists}$-smaller graph is generated and used in the lasso-finding tests (and only minimizations of graphs $\sqsubseteq^{\forall\exists}$-smaller than those in $G^f_{\mathcal{A}}$ are generated and used). The correctness of the algorithm is stated in Theorem 1, which is proved in [1] using the closure of $G^m_{\mathcal{A}}$ under composition stated in Lemma 8, the monotonicity from Lemma 6, and the preservation of lasso-finding tests from Lemma 7.

Theorem 1. *Algorithm 2 terminates. It returns* TRUE *iff $\mathcal{A}$ is universal.*

5 Language Inclusion of BA

Let $\mathcal{A} = (\Sigma, Q_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}}, \delta_{\mathcal{A}})$ and $\mathcal{B} = (\Sigma, Q_{\mathcal{B}}, I_{\mathcal{B}}, F_{\mathcal{B}}, \delta_{\mathcal{B}})$ be two BA. Let $\preceq_{\mathcal{A}}$ and $\preceq_{\mathcal{B}}$ be the maximal simulations on $\mathcal{A}$ and $\mathcal{B}$, respectively. We first introduce some further notations from [9] before explaining how to extend our approach from universality to language inclusion checking. Define the set $E_{\mathcal{A}} = Q_{\mathcal{A}} \times Q_{\mathcal{A}}$.

Algorithm 2. *Simulation-optimized Ramsey-based Universality Checking*

Input: A BA $\mathcal{A} = (\Sigma, Q, I, F, \delta)$, the set of all single-character graphs $G^1_{\mathcal{A}}$.
Output: TRUE if $\mathcal{A}$ is universal. Otherwise, FALSE.

```
1  Next := {Min(g) | g ∈ G^1_A}; Init := ∅;
2  while Next ≠ ∅ do
3      Pick and remove a graph g from Next;
4      if ∃f ∈ Init : f ⊑^{∀∃} g then Discard g and continue;
5      Remove all graphs f from Init s.t. g ⊑^{∀∃} f;
6      Add g into Init;
7  Next := Init; Processed := ∅;
8  while Next ≠ ∅ do
9      Pick a graph g from Next;
10     if ¬LFT(g, g) ∨ ∃h ∈ Processed : ¬LFT(h, g) ∨ ¬LFT(g, h) then
           return FALSE;
11     Remove g from Next and add it to Processed;
12     foreach h ∈ Init do
13         f = Min(g; h);
14         if ∃k ∈ Processed ∪ Next : k ⊑^{∀∃} f then Discard f and continue;
15         Remove all graphs k from Processed ∪ Next s.t. f ⊑^{∀∃} k;
16         Add f into Next;
17 return TRUE;
```

Each element in $E_{\mathcal{A}}$ is an edge $\langle p, q\rangle$ asserting that there is a path from p to q in $\mathcal{A}$. Define the language of an edge $\langle p, q\rangle$ as $\mathcal{L}(p, q) = \{w \in \Sigma^+ \mid p \overset{w}{\Longrightarrow} q\}$.

Define $S_{\mathcal{A},\mathcal{B}} = E_{\mathcal{A}} \times G_{\mathcal{B}}$. We call $\mathbf{g} = \langle \overline{g}, g\rangle$ a *supergraph* in $S_{\mathcal{A},\mathcal{B}}$. For any supergraph $\mathbf{g} \in S_{\mathcal{A},\mathcal{B}}$, its language $\mathcal{L}(\mathbf{g})$ is defined as $\mathcal{L}(\overline{g}) \cap \mathcal{L}(g)$. Let $Z_{\mathbf{gh}}$ be the ω-regular language $\mathcal{L}(\mathbf{g}) \cdot \mathcal{L}(\mathbf{h})^{\omega}$. We say $Z_{\mathbf{gh}}$ is *proper* if $\overline{g} = \langle p, q\rangle$ and $\overline{h} = \langle q, q\rangle$ for some $p \in I_{\mathcal{A}}$ and $q \in F_{\mathcal{A}}$. Notice that, by the definition of properness, every proper $Z_{\mathbf{gh}}$ is contained in $\mathcal{L}(\mathcal{A})$. The following is a relaxed version of Lemma 4 in [9] (the constraints of being a proper $Z_{\mathbf{gh}}$ are weaker than those in [9]).

Lemma 9. *(1)* $\mathcal{L}(\mathcal{A}) = \bigcup\{Z_{\mathbf{gh}} \mid Z_{\mathbf{gh}}$ *is proper*$\}$. *(2) For all non-empty proper* $Z_{\mathbf{gh}}$, *either* $Z_{\mathbf{gh}} \cap \mathcal{L}(\mathcal{B}) = \emptyset$ *or* $Z_{\mathbf{gh}} \subseteq \mathcal{L}(\mathcal{B})$. *(3)* $\mathcal{L}(\mathcal{A}) \cap \overline{\mathcal{L}(\mathcal{B})} = \bigcup\{Z_{\mathbf{gh}} \mid Z_{\mathbf{gh}}$ *is proper and* $Z_{\mathbf{gh}} \cap \mathcal{L}(\mathcal{B}) = \emptyset\}$.

The above lemma implies that $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff for all $\mathbf{g}, \mathbf{h} \in S_{\mathcal{A},\mathcal{B}}$, either $Z_{\mathbf{gh}}$ is not proper or $Z_{\mathbf{gh}} \subseteq \mathcal{L}(\mathcal{B})$. Since $\mathcal{L}(\mathbf{g}) = \emptyset$ or $\mathcal{L}(\mathbf{h}) = \emptyset$ implies $Z_{\mathbf{gh}} \subseteq \mathcal{L}(\mathcal{B})$, it is sufficient to build and check only supergraphs with nonempty languages (whose set we denote $S^f_{\mathcal{A},\mathcal{B}}$) when checking language inclusion.

Supergraphs in $S^f_{\mathcal{A},\mathcal{B}} = \{\mathbf{g} \in S_{\mathcal{A},\mathcal{B}} \mid \mathcal{L}(\mathbf{g}) \neq \emptyset\}$ can be built as follows. First, supergraphs $\mathbf{g} = \langle\langle p_{\mathbf{g}}, q_{\mathbf{g}}\rangle, g\rangle$ and $\mathbf{h} = \langle\langle p_{\mathbf{h}}, q_{\mathbf{h}}\rangle, h\rangle$ in $S_{\mathcal{A},\mathcal{B}}$ are *composable* iff $q_{\mathbf{g}} = p_{\mathbf{h}}$. Then, their composition $\mathbf{g};\mathbf{h}$ is defined as $\langle\langle p_{\mathbf{g}}, q_{\mathbf{h}}\rangle, g; h\rangle$. For all $a \in \Sigma$, define the set of single-character supergraphs $S^a = \{\langle\langle p, q\rangle, g_a\rangle \mid q \in \delta_{\mathcal{A}}(p, a)\}$. Let $S^1_{\mathcal{A},\mathcal{B}} := \bigcup_{a\in\Sigma} S^a$. As in universality checking, one can obtain

$S^f_{\mathcal{A},\mathcal{B}}$ by repeatedly composing composable supergraphs in $S^1_{\mathcal{A},\mathcal{B}}$ until a fixpoint is reached.

A method to check whether a pair of supergraphs $\langle \mathbf{g}, \mathbf{h} \rangle$ is a counterexample to $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$, i.e., a test whether $Z_{\mathbf{gh}}$ is both proper and disjoint from $\mathcal{L}(\mathcal{B})$, was proposed in [9]. A pair of supergraphs $\langle \mathbf{g} = \langle \overline{g}, g \rangle, \mathbf{h} = \langle \overline{h}, h \rangle \rangle$ passes the *double-graph test* (denoted $DGT(\mathbf{g}, \mathbf{h})$) iff $Z_{\mathbf{gh}}$ is not proper or $LFT(g, h)$. The following lemma is due to [9] .

Lemma 10. *$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff $DGT(\mathbf{g}, \mathbf{h})$ for all $\mathbf{g}, \mathbf{h} \in S^f_{\mathcal{A},\mathcal{B}}$.*

Analogously to the universality checking algorithm in Section 3, a language inclusion checking algorithm can be obtained by combining the above principles for generating supergraphs in $S^f_{\mathcal{A},\mathcal{B}}$ and using the double-graph test (cf. [1]).

6 Improving Language Inclusion Testing via Simulation

Here we describe our approach of utilizing simulation-based subsumption techniques to improve the Ramsey-based language inclusion test.

In order to be able to use simulation-based subsumption as in Section 4, we lift the subsumption relation $\sqsubseteq^{\forall\exists}$ to supergraphs as follows: Let $\mathbf{g} = \langle \langle p_{\mathbf{g}}, q_{\mathbf{g}} \rangle, g \rangle$ and $\mathbf{h} = \langle \langle p_{\mathbf{h}}, q_{\mathbf{h}} \rangle, h \rangle$ be two supergraphs in $S_{\mathcal{A},\mathcal{B}}$. Let $\mathbf{g} \sqsubseteq^{\forall\exists}_S \mathbf{h}$ iff $p_{\mathbf{g}} = p_{\mathbf{h}}$, $q_{\mathbf{g}} \succeq_{\mathcal{A}} q_{\mathbf{h}}$, and $g \sqsubseteq^{\forall\exists} h$. Define $\simeq^{\forall\exists}_S$ as $\sqsubseteq^{\forall\exists}_S \cap (\sqsubseteq^{\forall\exists}_S)^{-1}$.

Since we want to work with supergraphs that are minimal w.r.t. $\sqsubseteq^{\forall\exists}_S$, we need to change the definition of properness and the respective double-graph test. We say that $Z_{\mathbf{gh}}$ is *weakly proper* iff $\overline{g} = \langle p, q \rangle$ and $\overline{h} = \langle q_1, q_2 \rangle$ where $p \in I_{\mathcal{A}}$, $q_2 \in F_{\mathcal{A}}$, $q \succeq_{\mathcal{A}} q_1$, and $q_2 \succeq_{\mathcal{A}} q_1$.

Definition 3. *Supergraphs $\mathbf{g}, \mathbf{h} \in S_{\mathcal{A},\mathcal{B}}$ pass the relaxed double-graph test, written $RDGT(\mathbf{g}, \mathbf{h})$, iff either (1) $Z_{\mathbf{gh}}$ is not weakly proper, or (2) $LFT(g, h)$.*

The following Lemma 11 is needed to prove the central Theorem 2.

Lemma 11. *Every weakly proper $Z_{\mathbf{gh}}$ is contained in $\mathcal{L}(\mathcal{A})$.*

Theorem 2. *$\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$ iff $RDGT(\mathbf{g}, \mathbf{h})$ for all $\mathbf{g}, \mathbf{h} \in S^f_{\mathcal{A},\mathcal{B}}$.*

Furthermore, we lift the notions of $\leqslant^*$ and *Min* from Section 4 from graphs to supergraphs. For any two supergraphs $\mathbf{g} = \langle \overline{g}, g \rangle, \mathbf{h} = \langle \overline{h}, h \rangle$ from $S_{\mathcal{A},\mathcal{B}}$, we write $\mathbf{g} \leqslant^*_S \mathbf{h}$ iff $\overline{g} = \overline{h}$ and $g \leqslant^* h$. Then $S^m_{\mathcal{A},\mathcal{B}} = \{\mathbf{g} \in S_{\mathcal{A},\mathcal{B}} \mid \exists \mathbf{h} \in S^f_{\mathcal{A},\mathcal{B}} : \mathbf{g} \leqslant^*_S \mathbf{h}\}$. $Min_S(\mathbf{g})$ again computes a graph that is $\leqslant^*_S$-smaller than $\mathbf{g}$. It is a possibly non-deterministic operation such that $Min_S(\overline{g}, g) = \langle \overline{g}, Min(g) \rangle$.

Like in Section 4, it is now possible to prove a closure of $S^m_{\mathcal{A},\mathcal{B}}$ under composition as well as preservation of the double-graph test on $\sqsubseteq^{\forall\exists}_S$-larger supergraphs (cf. [1]). What slightly differs is the monotonicity of the composition, which is caused by the additional composability requirement. To cope with it, we define a new relation $\trianglelefteq^{\forall\exists}$, weakening $\sqsubseteq^{\forall\exists}_S$: For $\mathbf{g} = \langle \langle p, q \rangle, g \rangle, \mathbf{h} = \langle \langle p', q' \rangle, h \rangle \in S_{\mathcal{A},\mathcal{B}}$, $\mathbf{g} \trianglelefteq^{\forall\exists} \mathbf{h}$ iff $p' \preceq p$, $q' \preceq q$, and $g \sqsubseteq^{\forall\exists} h$. Notice that $\sqsubseteq^{\forall\exists}_S$ indeed implies $\trianglelefteq^{\forall\exists}$. From the definitions of $\sqsubseteq^{\forall\exists}_S$, $\trianglelefteq^{\forall\exists}$, and Lemma 6, we then easily get the following relaxed monotonicity lemma.

Algorithm 3. *Optimized Ramsey-based Language Inclusion Checking*

Input: BA $\mathcal{A} = (\Sigma, Q_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A}, \delta_\mathcal{A})$, $\mathcal{B} = (\Sigma, Q_\mathcal{B}, I_\mathcal{B}, F_\mathcal{B}, \delta_\mathcal{B})$, and the set $S^1_{\mathcal{A},\mathcal{B}}$.
Output: TRUE if $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. Otherwise, FALSE.

```
1  Next := {Min_S(g) | g ∈ S^1_{A,B}}; Init := ∅;
2  while Next ≠ ∅ do
3      Pick and remove a supergraph g from Next;
4      if ∃f ∈ Init : f ⊑^{∀∃}_S g then Discard g and continue;
5      Remove all supergraphs f from Init s.t. g ⊑^{∀∃}_S f;
6      Add g into Init;
7  Next := Init; Processed := ∅;
8  while Next ≠ ∅ do
9      Pick a supergraph g from Next;
10     if ¬RDGT(g, g) ∨ ∃h ∈ Processed : ¬RDGT(h, g) ∨ ¬RDGT(g, h) then
           return FALSE;
11     Remove g from Next and add it to Processed;
12     foreach h ∈ Init where ⟨g, h⟩ are composable do
13         f := Min_S(g; h);
14         if ∃k ∈ Processed ∪ Next : k ⊑^{∀∃}_S f then Discard f and continue;
15         Remove all supergraphs k from Processed ∪ Next s.t. f ⊑^{∀∃}_S k;
16         Add f into Next;
17 return TRUE;
```

Lemma 12. *For any* $\mathbf{e}, \mathbf{f}, \mathbf{g} \in S_{\mathcal{A},\mathcal{B}}$ *and* $\mathbf{h} \in S^m_{\mathcal{A},\mathcal{B}}$ *with* $\mathbf{e} \sqsubseteq^{\forall\exists}_S \mathbf{g}$, *and* $\mathbf{f} \trianglelefteq^{\forall\exists} \mathbf{h}$ *where* $\mathbf{e}, \mathbf{f}$ *and* $\mathbf{g}, \mathbf{h}$ *are composable,* $\mathbf{e}; \mathbf{f} \sqsubseteq^{\forall\exists}_S \mathbf{g}; \mathbf{h}$.

Now we show that it is safe to work with $\sqsubseteq^{\forall\exists}_S$-smaller supergraphs in the incremental supergraph construction. Given supergraphs $\mathbf{e}$, $\mathbf{g}$, $\mathbf{h}$ s.t. $\mathbf{e} \sqsubseteq^{\forall\exists}_S \mathbf{g}$ and $\mathbf{g}$, $\mathbf{h}$ are composable, one can always find a supergraph $\mathbf{f}$ satisfying the preconditions of Lemma 12—excluding the situation of only the bigger supergraphs $\mathbf{g}$, $\mathbf{h}$ being composable. Fortunately, it is possible to show that the needed supergraph $\mathbf{f}$ always exists in $S^m_{\mathcal{A},\mathcal{B}}$. Moreover, since only 1-letter supergraphs are used on the right of the composition, all supergraphs needed as right operands in the compositional construction can always be found in the minimization of $S^1_{\mathcal{A},\mathcal{B}}$, which can easily be generated.

Algorithm 3 is our simulation-optimized algorithm for BA inclusion-checking. Its correctness is stated below and proved in [1].

Theorem 3. *Algorithm 3 terminates. It returns* TRUE *iff* $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$.

7 Experimental Results

We have implemented both our simulation subsumption algorithms and the original ones of Fogarty and Vardi [9,10] in Java. For universality testing, we

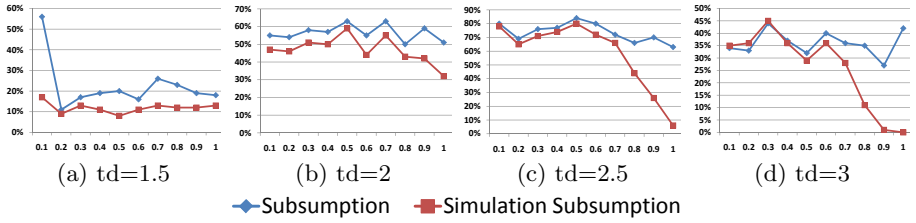


Fig. 1. Timeout cases of universality checking on Tabakov and Vardi's random model. The vertical axis is the percentage of tasks that cannot be finished within the timeout period and the horizontal axis is the acceptance density (ad).

compared our algorithm to the one in [10].[1] For language inclusion testing, we compared our simulation subsumption algorithm to a restricted version that uses the simple subsumption relation of [10]. (The language inclusion checking algorithm described in [9] does not use any subsumption at all and performed much worse.) We refer interested readers to [21] for all relevant materials needed to reproduce the results. A description of the machines that we used is given in [1].

Universality Testing. We have two sets of experiments. In Experiment 1, we use Tabakov and Vardi's random model. This model fixes the alphabet size to 2 and uses two parameters: *t*ransition *d*ensity (i.e. the average number of outgoing transitions per alphabet symbol) and *a*cceptance *d*ensity (percentage of accepting states). We use this approach with $td = 0.5, 1, \ldots, 4$ and $ad = 0.1, 0.2, \ldots, 1.0$, and generate 100 random BA for each combination of td and ad. In Figure 1, we compare the number of timeouts between the two approaches when the number of states is 100 and the timeout period is 3500 seconds[2]. We only list cases with $td = 1.5, 2.0, 2.5, 3.0$, since in the other cases both methods can complete most of the tasks within the timeout period. For the two most difficult configurations ($td = 2.5, 3.0$), the difference between the two approaches gets larger as ad increases. The trend shown in Figure 1 stayed the same when the number of states increases. We refer the interested readers to the Tech. Report [1] for results of automata with the number of states ranging from 10 to 200.

[1] The algorithm in [10] uses the simple subsumption relation $\sqsubseteq$, and also a different search strategy than our algorithm. To take into account the latter, we have also included a combination of our search strategy with the simple subsumption into our experiments. We evaluated the new search strategy with random automata of size 20. While our search strategy with the simple subsumption $\sqsubseteq$ is already on average about 20% faster than [10], the main improvement we witness in our experiments is achieved by using the simulation subsumption $\sqsubseteq^{\forall\exists}$.

[2] Our approach can be slightly slower than [9] in some rare cases, due to the overhead of computing simulation. For those cases, the simulation relation is sparse and does not yield any speedup. However, since there are efficient algorithms for computing simulation, the relative overhead is quite small on difficult instances.

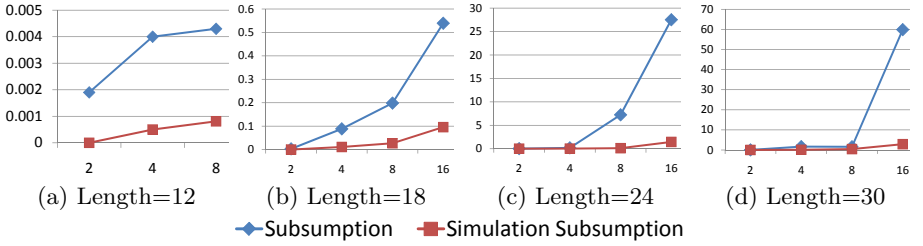


Fig. 2. Universality checking on automata generated from random LTL. Each point in the figure is the average result of 100 different instances. The horizontal axis is the size of the alphabet. The vertical axis is the average execution time in seconds.

Table 1. Language inclusion checking on mutual exclusion algorithms. The columns "Original" and "Error" refer to the original, resp., erroneous, model. We test inclusion for both directions. ">1day" = the task cannot be finished within one day. "oom" = required memory > 8GB. If a task completes successfully, we record the run time and the fact whether language inclusion holds or not ("T" or "F", respectively).

	Original		Error		Subsumption		Simulation Sub.	
	Tr.	St.	Tr.	St.	$\mathcal{L}(E)\subseteq\mathcal{L}(O)$	$\mathcal{L}(O)\subseteq\mathcal{L}(E)$	$\mathcal{L}(E)\subseteq\mathcal{L}(O)$	$\mathcal{L}(O)\subseteq\mathcal{L}(E)$
Bakery	2697	1506	2085	1146	>1day	>1day	32m15s(F)	49m57s
Peterson	34	20	33	20	2.7s(T)	1.4s(F)	0.9s(T)	0.2s(F)
Dining phil. Ver.1	212	80	464	161	>1day	>1day	11m52s(F)	>1day
Dining phil. Ver.2	212	80	482	161	>1day	>1day	14m40s(F)	>1day
Fisher Ver.1	1395	634	3850	1532	>1day	oom	1m23s(F)	>1day
Fisher Ver.2	1395	634	1420	643	>1day	>1day	8s(T)	8s(T)
MCS queue lock	21503	7963	3222	1408	>1day	>1day	1h36m(T)	6m22s(F)

Experiment 2 uses BA from random LTL formulae. We generate only valid formulae (in the form $f \vee \neg f$), thus ensuring that the corresponding BA are universal. We only focus on valid formulae since most BA generated from random LTL formulae can be recognized as non-universal in almost no time. Thus, the results would not have been interesting. We generate LTL formulae with lengths 12, 18, 24, and 30 and 1–4 propositions (which corresponds to automata with alphabet sizes 2, 4, 8, and 16). For each configuration, we generate 100 BA[3]. The results are shown in Figure 2. The difference between the two approaches is quite large in this set of experiments. With formulae of length 30 and 4 propositions, our approach is 20 times faster than the original subsumption based approach.

Language Inclusion Testing. We again have two sets of experiments. In Experiment 1, we inject (artificial) errors into models of several mutual exclusion algorithms from [15][4], translate both the original and the modified version into

[3] We do not have formulae having length 12 and 4 propositions because our generator [19] requires that $(\textit{length of formula}/3) > \textit{number of propositions}$.

[4] The models in [15] are based on guarded commands. We modify the models by randomly weakening or strengthening the guard of some commands.

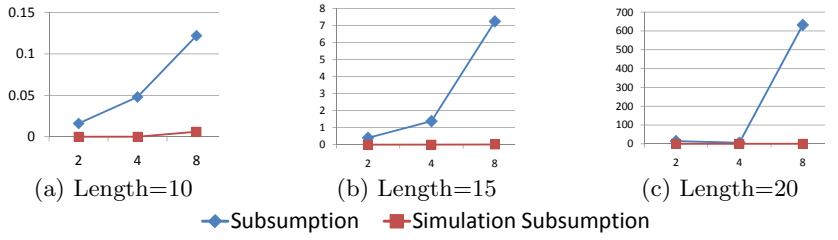


Fig. 3. Language inclusion checking on automata generated from random LTL. Each point in the figure is the average result of 10 different instances. The horizontal axis is the size of the alphabet. The vertical axis is the average execution time in seconds.

BA, and test whether the control flow (w.r.t. the occupation of the critical section) of the two versions is the same. In these models, a state p is accepting iff the critical section is occupied by some process in p. The results are shown in Table 1. The results of a slightly different version where all states are accepting is given in [1]. In both versions, our approach has significantly and consistently better performance than the basic subsumption approach.

In Experiment 2, we translate two randomly generated LTL formulae to BA $\mathcal{A}$, $\mathcal{B}$ and then check whether $\mathcal{L}(\mathcal{A}) \subseteq \mathcal{L}(\mathcal{B})$. We use formulae having length 10, 15, and 20 and 1–3 propositions (which corresponds to BA of alphabet sizes 2, 4, and 8). For each length of formula and each number of propositions, we generate 10 pairs of BA. The relative results are shown in Figure 3. The difference in performance of using the basic subsumption and the simulation subsumption is again quite significant here. For the largest cases (with formulae having length 20 and 3 propositions), our approach is 1914 times faster than the basic subsumption approach.

8 Conclusions and Extensions

We have introduced simulation-based subsumption techniques for Ramsey-based universality/language-inclusion checking for nondeterministic BA. We evaluated our approach by a wide set of experiments, showing that the simulation-subsumption approach consistently outperforms the basic subsumption of [10].

Our techniques can be extended in several ways. Weaker simulations for BA have been defined in the literature, e.g., delayed/fair simulation [7], or their multipebble extensions [6]. One possibility is to quotient the BA w.r.t. (multipebble) delayed simulation, which (unlike quotienting w.r.t. fair simulation) preserves the language. Furthermore, in our language inclusion checking algorithm, the subsumption w.r.t. direct simulation $\preceq_{\mathcal{A}}$ on $\mathcal{A}$ can be replaced by the weaker delayed simulation (but not by fair simulation). Moreover, in the definition of being *weakly proper* in Section 6, in the condition $q \succeq_{\mathcal{A}} q_1$, the $\succeq_{\mathcal{A}}$ can be replaced by any other relation that implies ω-language containment, e.g., fair simulation. On the other hand, delayed/fair simulation cannot be used for subsumption in the automaton $\mathcal{B}$ in inclusion checking (nor in universality checking), since Lemma 6 does not carry over.

Next, our language-inclusion algorithm does not currently exploit any simulation preorders *between* $\mathcal{A}$ and $\mathcal{B}$. Of course, direct/delayed/fair simulation between the respective initial states of $\mathcal{A}$ and $\mathcal{B}$ is sufficient (but not necessary) for language inclusion. However, it is more challenging to exploit simulation preorders between internal states of $\mathcal{A}$ and internal states of $\mathcal{B}$.

Finally, it is easy to see that the proposed simulation subsumption technique can be built over *backward simulation preorder* too. It would, however, be interesting to evaluate such an approach experimentally. Further, one could then also try to extend the framework to use some form of *mediated preorders* combining forward and backward simulation preorders like in the approach of [3].

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. Technical report FIT-TR-2010-02, FIT BUT (2010), http://www.fit.vutbr.cz/~holik/pub/FIT-TR-2010-002.pdf
2. Abdulla, P.A., Chen, Y.-F., Holík, L., Mayr, R., Vojnar, T.: When Simulation Meets Antichains (On Checking Language Inclusion of Nondeterministic Finite (Tree) Automata). In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 158–174. Springer, Heidelberg (2010)
3. Abdulla, P.A., Chen, Y.-F., Holík, L., Vojnar, T.: Mediating for Reduction (On Minimizing Alternating Büchi Automata). In: Proc. of FSTTCS'09, Leibniz International Proceedings in Informatics, vol. 4 (2009)
4. Büchi, J.R.: On a Decision Method in Restricted Second Order Arithmetic. In: Proc. of Int. Con. on Logic, Method, and Phil. of Science (1962)
5. Doyen, L., Raskin, J.-F.: Improved Algorithms for the Automata-based Approach to Model Checking. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 451–465. Springer, Heidelberg (2007)
6. Etessami, K.: A Hierarchy of Polynomial-Time Computable Simulations for Automata. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, p. 131. Springer, Heidelberg (2002)
7. Etessami, K., Wilke, T., Schuller, R.A.: Fair Simulation Relations, Parity Games, and State Space Reduction for Büchi Automata. SIAM J. Comp. 34(5) (2005)
8. Fogarty, S.: Büchi Containment and Size-Change Termination. Master's Thesis, Rice University (2008)
9. Fogarty, S., Vardi, M.Y.: Büchi Complementation and Size-Change Termination. In: Proc. of TACAS'09. LNCS, vol. 5505. Springer, Heidelberg (2009)
10. Fogarty, S., Vardi, M.Y.: Efficient Büchi Universality Checking. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 205–220. Springer, Heidelberg (2010)
11. Henzinger, M.R., Henzinger, T.A., Kopke, P.W.: Computing Simulations on Finite and Infinite Graphs. In: Proc. FOCS'95. IEEE CS, Los Alamitos (1995)
12. Holík, L., Šimáček, J.: Optimizing an LTS-Simulation Algorithm. In: Proc. of MEMICS'09 (2009)
13. Jones, N.D., Lee, C.S., Ben-Amram, A.M.: The Size-Change Principle for Program Termination. In: Proc. of POPL'01. ACM SIGPLAN (2001)
14. Kupferman, O., Vardi, M.Y.: Weak Alternating Automata Are Not That Weak. ACM Transactions on Computational Logic 2(2), 408–429 (2001)

15. 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)
16. Sistla, A.P., Vardi, M.Y., Wolper, P.: The Complementation Problem for Büchi Automata with Applications to Temporal Logic. In: Brauer, W. (ed.) ICALP 1985. LNCS, vol. 194. Springer, Heidelberg (1985)
17. Somenzi, F., Bloem, R.: Efficient Büchi Automata from LTL Formulae. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855. Springer, Heidelberg (2000)
18. Tabakov, D., Vardi, M.Y.: Model Checking Büchi Specifications. In: Proc. of LATA'07 (2007)
19. 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)
20. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.-F.: Antichains: A New Algorithm for Checking Universality of Finite Automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
21. http://iis.sinica.edu.tw/FMLAB/CAV2010
(capitalize "FMLAB" and "CAV 2010")

Efficient Emptiness Check for Timed Büchi Automata

Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz

LaBRI (Université de Bordeaux -CNRS)

Abstract. The Büchi non-emptiness problem for timed automata concerns deciding if a given automaton has an infinite non-Zeno run satisfying the Büchi accepting condition. The standard solution to this problem involves adding an auxiliary clock to take care of the non-Zenoness. In this paper, we show that this simple transformation may sometimes result in an exponential blowup. We propose a method avoiding this blowup.

1 Introduction

Timed automata [1] are widely used to model real-time systems. They are obtained from finite automata by adding clocks that can be reset and whose values can be compared with constants. The crucial property of timed automata is that their emptiness is decidable. This model has been implemented in verification tools like Uppaal [3] or Kronos [7], and used in industrial case studies [12,4,13].

While most tools concentrate on the reachability problem, the questions concerning infinite executions of timed automata are also of interest. In the case of infinite executions one has to eliminate so called Zeno runs. These are executions that contain infinitely many steps taken in a finite time interval. For obvious reasons such executions are considered unrealistic. In this paper we study the problem of deciding if a given timed automaton has a non-Zeno run passing through accepting states infinitely often. We call this problem *Büchi non-emptiness* problem.

This basic problem has been of course studied already in the paper introducing timed automata. It has been shown that using so called region abstraction the problem can be reduced to the problem of finding a path in a finite region graph satisfying some particular conditions. The main difference between the cases of finite and infinite executions is that in the latter one needs to decide if the path that has been found corresponds to a non Zeno run of the automaton.

Subsequent research has shown that the region abstraction is very inefficient for reachability problems. Another method using zones instead of regions has been proposed. It is used at present in all timed-verification tools. While simple at the first sight, the zone abstraction was delicate to get right. This is mainly because the basic properties of regions do not transfer to zones. The zone abstraction also works for infinite executions, but unlike for regions, it is impossible to decide if a path in a zone graph corresponds to a non-Zeno run of the automaton.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 148–161, 2010.

There exists a simple solution to the problem of Zeno runs that amounts to transforming automata in such way that every run passing through an accepting state infinitely often is non-Zeno. An automaton with such a property is called *strongly non-Zeno*. The transformation is easy to describe and requires addition of one new clock. This paper is motivated by our experiments with an implementation of this construction. We have observed that this apparently simple transformation can give a big overhead in the size of a zone graph.

In this paper we closely examine the transformation to strongly non-Zeno automata [17], and show that it can inflict a blowup of the zone graph; and this blowup could even be exponential in the number of clocks. To substantiate, we exhibit an example of an automaton having a zone graph of polynomial size, whose transformed version has a zone graph of exponential size. We propose another solution to avoid this phenomenon. Instead of modifying the automaton, we modify the zone graph. We show that this modification allows us to detect if a path can be instantiated to a non-Zeno run. Moreover the size of the modified graph is $|ZG(\mathcal{A})|.|X|$, where $|ZG(\mathcal{A})|$ is the size of the zone graph and $|X|$ is the number of clocks.

In the second part of the paper we propose an algorithm for testing the existence of accepting non-Zeno runs in timed automata. The problem we face highly resembles the emptiness testing of finite automata with generalized Büchi conditions. Since the most efficient solutions for the latter problem are based on the Tarjan's algorithm, we take the same way here. We present an algorithm whose running time is bounded by $|ZG(\mathcal{A})|.|X|^2$. We also report on the experiments performed with a preliminary implementation of this algorithm.

Related work. The zone approach has been introduced in Petri net context [5], and then adapted to the framework of timed automata [9]. The advantage of zones over regions is that they do not require to consider every possible unit time interval separately. The delicate point about zones was to find a right approximation operator. Indeed while regions are both pre- and post-stable, zones are not pre-stable, and some care is needed to guarantee post-stability. Post-stability is enough for correctness of the reachability algorithm, and for testing if a path in the zone graph can be instantiated to a run of the automaton. It is however not possible to determine if a path can be instantiated to a non-Zeno run. The solution involving adding one clock has been discussed in [15,17,2]. Recently, Tripakis [16] has shown a way to extract an accepting run from a zone graph of the automaton. Combined with the construction of adding one clock this gives a solution to our problem. A different approach has been considered in [11] where syntactic conditions are proposed for a timed automaton to be free from Zeno runs. Notice that for obvious complexity reasons, any such condition must be either not complete, or of the same algorithmic complexity as the emptiness test itself.

Organization of the paper. In the next section we formalize our problem, and discuss region and zone abstractions. As an intermediate step we give a short proof of the above mentioned result from [16]. Section 3 explains the problems with the transformation to strongly non-Zeno automata, and describes our alternative method. The following section is devoted to a description of the algorithm.

2 The Emptiness Problem for Timed Büchi Automata

2.1 Timed Büchi Automata

Let X be a set of clocks, i.e., variables that range over $\mathbb{R}_{\geq 0}$, the set of non-negative real numbers. *Clock constraints* are conjunctions of comparisons of variables with integer constants, 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 $\nu : X \to \mathbb{R}_{\geq 0}$. We denote $\mathbb{R}_{\geq 0}^X$ for the set of clock valuations over X, and $\mathbf{0} : X \to \{0\}$ for the valuation that associates 0 to every clock in X. We write $\nu \models \phi$ when ν satisfies ϕ, i.e. when every constraint in ϕ holds after replacing every x by $\nu(x)$.

For a valuation ν and $\delta \in \mathbb{R}_{\geq 0}$, let $(\nu + \delta)$ be the valuation such that $(\nu + \delta)(x) = \nu(x) + \delta$ for all $x \in X$. For a set $R \subseteq X$, let $[R]\nu$ be the valuation such that $([R]\nu)(x) = 0$ if $x \in R$ and $([R]\nu)(x) = \nu(x)$ otherwise.

A *Timed Büchi Automaton (TBA)* 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 a *reset* of the transition.

A *configuration* of $\mathcal{A}$ is a pair $(q, \nu) \in Q \times \mathbb{R}_{\geq 0}^X$; with $(q_0, \mathbf{0})$ being the *initial configuration*. A *discrete transition* between configurations $(q, \nu) \xrightarrow{t} (q', \nu')$ for $t = (q, g, R, q')$ is defined when $\nu \vDash g$ and $\nu' = [R]\nu$. We also have *delay transitions* between configurations: $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta)$ for every q, ν and $\delta \in \mathbb{R}_{\geq 0}$. We write $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$ if $(q, \nu) \xrightarrow{\delta} (q, \nu + \delta) \xrightarrow{t} (q', \nu')$.

A *run* of $\mathcal{A}$ is a finite or infinite sequence of configurations connected by $\xrightarrow{\delta, t}$ transitions, starting from the initial state q_0 and the initial valuation $\nu_0 = \mathbf{0}$:

$$(q_0, \nu_0) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots$$

A run σ *satisfies the Büchi condition* if it visits *accepting configurations* infinitely often, that is configurations with a state from Acc. The *duration* of the run is the accumulated delay: $\sum_{i \geq 0} \delta_i$. A run σ is *Zeno* if its duration is bounded.

Definition 1. *The* Büchi non-emptiness problem *is to decide if $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition.*

The class of TBA we consider is usually known as diagonal-free TBA since clock comparisons like $x - y \leq 1$ are disallowed. Since we are interested in the Büchi non-emptiness problem, we can consider automata without an input alphabet and without invariants since they can be simulated by guards.

The Büchi non-emptiness problem is known to be PSPACE-complete [1].

2.2 Regions and Region Graphs

A simple decision procedure for the Büchi non-emptiness problem builds from $\mathcal{A}$ a graph called the *region graph* and tests if there is a path in this graph satisfying certain conditions. We will define two types of regions.

Fix a constant M and a finite set of clocks X. Two valuations $\nu, \nu' \in \mathbb{R}^X_{\geq 0}$ are *region equivalent* w.r.t. M, denoted $\nu \sim_M \nu'$ iff for every $x, y \in X$:

1. $\nu(x) > M$ iff $\nu'(x) > M$;
2. if $\nu(x) \leq M$, then $\lfloor \nu(x) \rfloor = \lfloor \nu'(x) \rfloor$;
3. if $\nu(x) \leq M$, then $\{\nu(x)\} = 0$ iff $\{\nu'(x)\} = 0$;
4. if $\nu(x) \leq M$ and $\nu(y) \leq M$ then $\{\nu(x)\} \leq \{\nu(y)\}$ iff $\{\nu'(x)\} \leq \{\nu'(y)\}$.

The first three conditions ensure that the two valuations satisfy the same guards. The last one enforces that for every $\delta \in \mathbb{R}_{\geq 0}$ there is $\delta' \in \mathbb{R}_{\geq 0}$, such that valuations $\nu + \delta$ and $\nu' + \delta'$ satisfy the same guards.

We will also define *diagonal region equivalence (d-region equivalence* for short) that strengthens the last condition to

4^d**.** for every integer $c \in (-M, M)$: $\nu(x) - \nu(y) \leq c$ iff $\nu'(x) - \nu'(y) \leq c$

This region equivalence is denoted by $\sim^d_M$. Observe that it is finer than $\sim_M$.

A *region* is an equivalence class of $\sim_M$. We write $[\nu]_{\sim_M}$ for the region of ν, and $\mathcal{R}_M$ for the set of all regions with respect to M. Similarly, for d-region equivalence we write: $[\nu]^d_{\sim_M}$ and $\mathcal{R}^d_M$. If r is a region or a d-region then we will write $r \vDash g$ to mean that every valuation in r satisfies the guard g. Observe that all valuations in a region, or a d-region, satisfy the same guards.

For an automaton $\mathcal{A}$, we define its *region graph*, $RG(\mathcal{A})$, using $\sim_M$ relation, where M is the biggest constant appearing in the guards of its transitions. Nodes of $RG(\mathcal{A})$ are of the form (q, r) for q a state of $\mathcal{A}$ and $r \in \mathcal{R}_M$ a region. There is a transition $(q, r) \xrightarrow{t} (q', r')$ if there are $\nu \in r$, $\delta \in \mathbb{R}_{\geq 0}$ and $\nu' \in r'$ with $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$. Observe that a transition in the region graph is not decorated with a delay. The graph $RG^d(\mathcal{A})$ is defined similarly but using the $\sim^d_M$ relation.

It will be important to understand the properties of pre- and post-stability of regions or d-regions [17]. We state them formally. A transition $(q, r) \xrightarrow{t} (q', r')$ in a region graph or a d-region graph is:

- *Pre-stable* if for every $\nu \in r$ there are $\nu' \in r'$, $\delta \in \mathbb{R}_{\geq 0}$ s.t. $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$.
- *Post-stable* if for every $\nu' \in r'$ there are $\nu \in r$, $\delta \in \mathbb{R}_{\geq 0}$ s.t. $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$.

The following lemma (cf. [6]) explains our interest in $\sim^d_M$ relation.

Lemma 1 (Pre and post-stability). *Transitions in $RG^d(\mathcal{A})$ are pre-stable and post-stable. Transitions in $RG(\mathcal{A})$ are pre-stable but not necessarily post-stable.*

Consider two sequences

$$(q_0, \nu_0) \xrightarrow{\delta_0, t_0} (q_1, \nu_1) \xrightarrow{\delta_1, t_1} \cdots \tag{1}$$

$$(q_0, r_0) \xrightarrow{t_0} (q_1, r_1) \xrightarrow{t_1} \cdots \tag{2}$$

where the first is a run in $\mathcal{A}$, and the second is a path in $RG(\mathcal{A})$ or $RG^d(\mathcal{A})$. We say that the first is an *instance* of the second if $\nu_i \in r_i$ for all $i \geq 0$. Equivalently, we say that the second is an *abstraction* of the first. The following lemma is a direct consequence of the pre-stability property.

Lemma 2. *Every path in $RG(\mathcal{A})$ is an abstraction of a run of $\mathcal{A}$, and conversely, every run of $\mathcal{A}$ is an instance of a path in $RG(\mathcal{A})$. Similarly for $RG^d(\mathcal{A})$.*

This lemma allows us to relate the existence of an accepting run of $\mathcal{A}$ to the existence of paths with special properties in $RG(\mathcal{A})$ or $RG^d(\mathcal{A})$. We say that a path as in (2) *satisfies the Büchi condition* if it has infinitely many occurrences of states from Acc. The path is called *progressive* if for every clock $x \in X$:

- either x is almost always above M: there is n with $r_i \vDash x > M$ for all $i > n$;
- or x is reset infinitely often and strictly positive infinitely often: for every n there are $i, j > n$ such that $r_i \vDash (x = 0)$ and $r_j \vDash (x > 0)$.

Theorem 1 ([1]). *For every TBA $\mathcal{A}$, $\mathcal{L}(\mathcal{A}) \neq \emptyset$ iff $RG(\mathcal{A})$ has a progressive path satisfying the Büchi condition. Similarly for $RG^d(\mathcal{A})$.*

While this theorem gives an algorithm for solving our problem, it turns out that this method is very impractical. The number of regions $\mathsf{RA}(\mathcal{A})$ is $\mathcal{O}(|X|!2^{|X|}M^{|X|})$ [1] and constructing all of them, or even searching through them on-the-fly, has proved to be very costly.

2.3 Zones and Zone Graphs

Timed verification tools use zones instead of regions. A zone is a set of valuations defined by a conjunction of two kinds of constraints : comparison of the difference between two clocks with a constant, or comparison of the value of a single clock with a constant. For example $(x - y \geq 1) \wedge (y < 2)$ is a zone. While at first sight it may seem that there are more zones than regions, this is not the case if we count only those that are reachable from the initial valuation.

Since zones are sets of valuations defined by constraints, one can define discrete and delay transitions directly on zones. For $\delta \in \mathbb{R}_{\geq 0}$, we have $(q, Z) \xrightarrow{\delta} (q, Z')$ if Z' is the smallest zone containing the set of all the valuations $\nu + \delta$ with $\nu \in Z$. Similarly, for a discrete transition we put $(q, Z) \xrightarrow{t} (q', Z')$ if Z' is the set of all the valuations ν' such that $(q, \nu) \xrightarrow{t} (q', \nu')$ for some $\nu \in Z$; Z' is a zone in this case. Moreover zones can be represented using Difference Bound Matrices (DBMs), and transitions can be computed efficiently on DBMs [9]. The problem is that the number of reachable zones is not guaranteed to be finite [8].

In order to ensure that the number of reachable zones is finite, one introduces abstraction operators. We mention the three most common ones in the literature. They refer to region graphs, $RG(\mathcal{A})$ or $RG^d(\mathcal{A})$, and use the constant M that is the maximal constant appearing in the guards of $\mathcal{A}$.

- $Closure_M(Z)$: the smallest union of regions containing Z;
- $Closure^d_M(Z)$: similarly but for d-regions;
- $Approx_M(Z)$: the smallest union of d-regions that is convex and contains Z.

The following lemma establishes the links between the three abstraction operators, and is very useful to transpose reachability results from one abstraction to the other.

Lemma 3 ([6]). *For every zone Z: $Z \subseteq Closure^d_M(Z) \subseteq Approx_M(Z) \subseteq Closure_M(Z)$.*

A symbolic zone S is a zone approximated with one of the above abstraction operators. Now, similar to region graphs, we define simulation graphs where after every transition a specific approximation operation is used, that is each node in the simulation graph is of the form (q, S) with S being a symbolic zone . So we have three graphs corresponding to the three approximation operations.

Take an automaton $\mathcal{A}$ and let M be the biggest constant appearing in the guards of its transitions. In the simulation graph $SG(\mathcal{A})$ the nodes are of the form $(q, Closure_M(Z))$ where q is a state of $\mathcal{A}$ and Z is a zone. The initial node is $(q_0, Closure_M(Z_0))$, with q_0 the initial state of $\mathcal{A}$, and Z_0 the zone setting all the variables to 0. The transitions in the graph are $(q, Closure_M(Z)) \xrightarrow{t} (q', Closure_M(Z'))$ where Z' is the set of valuations ν' such that there exist $\nu \in Closure_M(Z)$ and $\delta \in \mathbb{R}_{\geq 0}$ satisfying $(q, \nu) \xrightarrow{\delta, t} (q', \nu')$. Similarly for $SG^d(\mathcal{A})$ and $SG^a(\mathcal{A})$ but replacing $Closure_M$ with operations $Closure^d_M$ and $Approx_M$, respectively. The notions of an abstraction of a run of $\mathcal{A}$, and an instance of a path in a simulation graph are defined in the same way as that of region graphs.

Tools like Kronos or Uppaal use $Approx_M$ abstraction. The two others are less interesting for implementations since the result may not be convex. Nevertheless, they are useful in proofs. The following lemma (cf. [8]) says that transitions in $SG(\mathcal{A})$ are post-stable with respect to regions.

Lemma 4. *Let $(q, S) \xrightarrow{t} (q', S')$ be a transition in $SG(\mathcal{A})$. For every region $r' \subseteq S'$, there is a region $r \subseteq S$ such that $(q, r) \xrightarrow{t} (q', r')$ is a transition in $RG(\mathcal{A})$.*

We get a correspondence between paths in simulation graphs and runs of $\mathcal{A}$.

Theorem 2 ([16]). *Every path in $SG(\mathcal{A})$ is an abstraction of a run of $\mathcal{A}$, and conversely, every run of $\mathcal{A}$ is an instance of a path in $SG(\mathcal{A})$. Similarly for SG^d and SG^a.*

Proof. We first show that a path in $SG(\mathcal{A})$ is an abstraction of a run of $\mathcal{A}$. Take a path $(q_0, S_0) \xrightarrow{t_0} (q_1, S_1) \xrightarrow{t_1} \dots$ in $SG(\mathcal{A})$. Construct a DAG with nodes (i, q_i, r_i) such that r_i is a region in S_i. We put an edge from (i, q_i, r_i) to $(i+1, q_{i+1}, r_{i+1})$ if $(q_i, r_i) \xrightarrow{t_i} (q_{i+1}, r_{i+1})$. By Lemma 4, every node in this DAG has at least one predecessor, and the branching of every node is bounded by the number of regions. Hence, this DAG has an infinite path that is a path in $RG(\mathcal{A})$. By Lemma 2 this path can be instantiated to a run of $\mathcal{A}$.

To conclude the proof one shows that a run of $\mathcal{A}$ can be abstracted to a path in $SG^d(\mathcal{A})$. Then using Lemma 3 this path can be converted to a path in $SG^a(\mathcal{A})$, and later to one in $SG(\mathcal{A})$. We omit the details.

Observe that this theorem does not guarantee that a path we find in a simulation graph has an instantiation that is non-Zeno. It is indeed impossible to guarantee this unless some additional conditions on paths or automata are imposed.

In the subsequent sections, we are interested only in the simulation graph SG^a. Observe that the symbolic zone obtained by the approximation of a zone using $Approx_M$ is in fact a zone. Hence, we prefer to call it a zone graph and denote it as ZG^a. Every node of ZG^a is of the form (q, Z) where Z is a zone.

3 Finding Non Zeno Paths

As we have remarked above, in order to use Theorem 2 we need to be sure that a path we get can be instantiated to a non-Zeno run. We discuss the solutions proposed in the literature, and then offer a better one. Thanks to pre-stability of the region graph, the progress criterion on regions has been defined in [1] for selecting runs from $RG(\mathcal{A})$ that have a non-Zeno instantiation (see Section 2.2). Notice that the semantics of TBA in [1] constrains all delays δ_i to be strictly positive, but the progress criterion can be extended to the stronger semantics that is used nowadays (see [17] for instance). However, since zone graphs are not pre-stable, this method cannot be directly extended to zone graphs.

3.1 Adding One Clock

A common solution to deal with Zeno runs is to transform an automaton into a *strongly non-Zeno automaton*, i.e. such that all runs satisfying the Büchi condition are guaranteed to be non-Zeno. We present this solution here and discuss why, although simple, it may add an exponential factor in the decision procedure.

The transformation of $\mathcal{A}$ into a strongly non-Zeno automaton $SNZ(\mathcal{A})$ proposed in [17] adds one clock z and duplicates accepting states. One copy of the state is as before but is no longer accepting. The other copy is accepting, but it can be reached only when $z \geq 1$. Moreover when it is reached z is reset to 0. The only transition from this second copy leads to the first copy. (See $\mathcal{V}_k$ and $\mathcal{W}_k$ on Figure 1 for an example.) This construction ensures that at least one unit of time has passed between two visits to an accepting state. A slightly different construction is mentioned in [2]. Of course one can also have other modifications, and it is impossible to treat all the imaginable constructions at once. Our objective here is to show that the constructions proposed in the literature produce a phenomenon that causes proliferation of zones that can sometimes be exponential in the number of clocks. The discussion below will focus on the construction from [17], but the one from [2] suffers from the same problem.

The problem comes from the fact that the constraint $z \geq 1$ may be a source of rapid multiplication of the number of zones in the zone graph of $SNZ(\mathcal{A})$. Consider $\mathcal{V}_k$ and $\mathcal{W}_k$ from Figure 1 for $k = 2$. Starting at the state b^2 of $\mathcal{V}_2$ in the zone $0 \leq y \leq x_1 \leq x_2$, there are two reachable zones with state b^2. Moreover, if we reset x_1 followed by y from the two zones, we reach the same zone $0 \leq y \leq x_1 \leq x_2$. In contrast starting in b^2 of $\mathcal{W}_2 = SNZ(\mathcal{V}_2)$ from $0 \leq y \leq x_1 \leq x_2 \leq z$ gives at least d zones, and resetting x_1 followed by y still yields d zones.

We now exploit this fact to give an example of a TBA $\mathcal{A}_n$ whose zone graph has a number of zones linear in the number of clocks, but $\mathcal{B}_n = SNZ(\mathcal{A}_n)$ has a

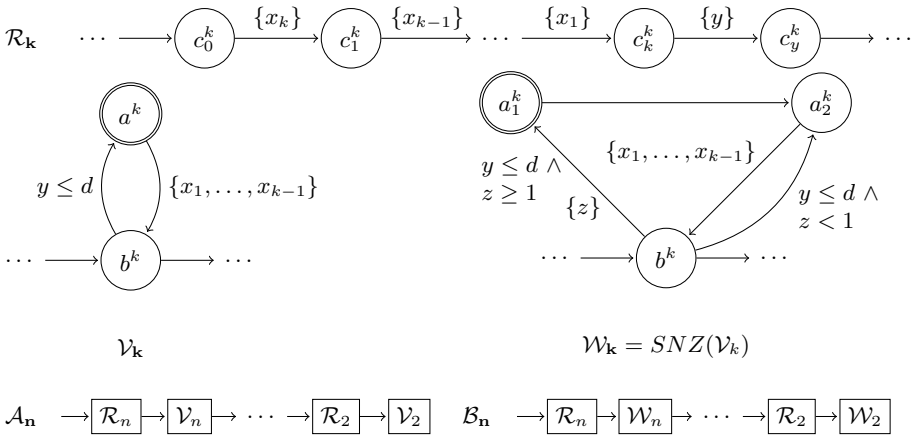


Fig. 1. Gadgets for $\mathcal{A}_n$ and $\mathcal{B}_n = SNZ(\mathcal{A}_n)$

zone graph of size exponential in the number of clocks. $\mathcal{A}_n$ is constructed from the automata gadgets $\mathcal{V}_k$ and $\mathcal{R}_k$ as shown in Figure 1. Observe that the role of $\mathcal{R}_k$ is to enforce an order $0 \le y \le x_1 \le \cdots \le x_k$ between clock values. By induction on k one can compute that there are only two zones at locations b^k since $\mathcal{R}_{k+1}$ made the two zones in b^{k+1} collapse into the same zone in b^k. Hence the number of nodes in the zone graph of $\mathcal{A}_n$ is $\mathcal{O}(n)$.

Let us now consider $\mathcal{B}_n$, the strongly non-Zeno automaton obtained from $\mathcal{A}_n$ following [17]. Every gadget $\mathcal{V}_k$ gets transformed to $\mathcal{W}_k$. While exploring $\mathcal{W}_k$, one introduces a distance between the clocks x_{k-1} and x_k. So when leaving it one gets zones with $x_k - x_{k-1} \ge c$, where $c \in \{0, 1, 2, \ldots, d\}$. The distance between x_k and x_{k-1} is preserved by $\mathcal{R}_k$. In consequence, $\mathcal{W}_n$ produces at least $d+1$ zones. For each of these zones $\mathcal{W}_{n-1}$ produces $d+1$ more zones. In the end, the zone graph of $\mathcal{B}_n$ has at least $(d+1)^{n-1}$ zones at the state b^2.

We have thus shown that $\mathcal{A}_n$ has $\mathcal{O}(n)$ zones while $\mathcal{B}_n = SNZ(\mathcal{A}_n)$ has an exponential number of zones even when the constant d is 1. Observe that the construction shows that even with two clocks the number of zones blows exponentially in the binary representation of d. Note that the automaton $\mathcal{A}_n$ does not have a non-Zeno accepting run. Hence, every search algorithm is compelled to explore all the zones of $\mathcal{B}_n$.

3.2 A More Efficient Solution

Our solution stems from a realization that we only need one non-Zeno run satisfying the Büchi condition and so in a way transforming an automaton to strongly non-Zeno is an overkill. We propose not to modify the automaton, but to introduce additional information to the zone graph $ZG^a(\mathcal{A})$. The nodes will now be triples (q, Z, Y) where $Y \subseteq X$ is the set of clocks that can potentially be equal to 0. It means in particular that other clock variables, i.e. those from $X - Y$ are

assumed to be bigger than 0. We write $(X - Y) > 0$ for the constraint saying that all the variables in $X - Y$ are not 0.

Definition 2. *Let $\mathcal{A}$ be a TBA over a set of clocks X. The* guessing zone graph *$GZG^a(\mathcal{A})$ has nodes of the form (q, Z, Y) where (q, Z) is a node in $ZG^a(\mathcal{A})$ and $Y \subseteq X$. The initial node is (q_0, Z_0, X), with (q_0, Z_0) the initial node of $ZG^a(\mathcal{A})$. There is a transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ in $GZG^a(\mathcal{A})$ if there is a transition $(q, Z) \xrightarrow{t} (q', Z')$ in $ZG^a(\mathcal{A})$ with $t = (q, g, R, q')$, and there are valuations $\nu \in Z$, $\nu' \in Z'$, and δ such that $\nu + \delta \models (X - Y) > 0$ and $(q, \nu) \xrightarrow{\delta,t} (q, \nu')$. We also introduce a new auxiliary letter τ, and put transitions $(q, Z, Y) \xrightarrow{\tau} (q, Z, Y')$ for $Y' = \emptyset$ or $Y' = Y$.*

Observe that the definition of transitions reflects the intuition about Y we have described above. Indeed, the additional requirement on the transition $(q, Z, Y) \xrightarrow{t} (q', Z', Y \cup R)$ is that it should be realizable when the clocks outside Y are strictly positive; so there should be a valuation satisfying $(X - Y) > 0$ that realizes this transition. As we will see later, this construction entails that from a node $(q, Z, \emptyset)$ every reachable zero-check is preceded by the reset of the variable that is checked, and hence nothing prevents a time elapse in this node. A node of the form $(q, Z, \emptyset)$ is called *clear*. We call a node (q, Z, Y) *accepting* if it contains an accepting state q.

Example. Figure 2 depicts a TBA $\mathcal{A}_1$ along with its zone graph $ZG^a(\mathcal{A}_1)$ and its guessing zone graph $GZG^a(\mathcal{A}_1)$ where τ-loops have been omitted.

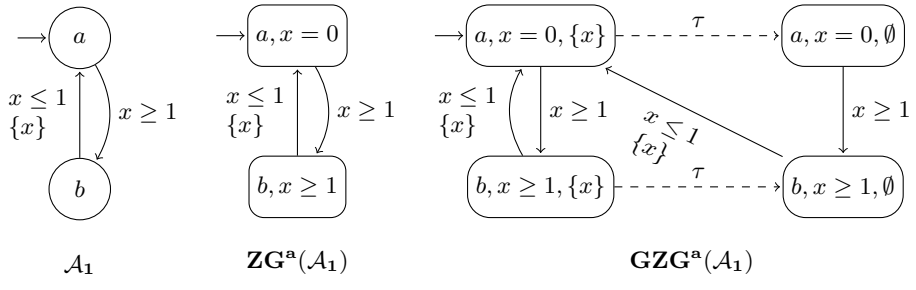


Fig. 2. A TBA $\mathcal{A}_1$ and the guessing zone graph $GZG^a(\mathcal{A}_1)$

Notice that directly from the definition it follows that a path in $GZG^a(\mathcal{A})$ determines a path in $ZG^a(\mathcal{A})$ obtained by removing τ transitions and the third component from nodes.

A variable x is *bounded* by a transition of $GZG^a(\mathcal{A})$ if the guard of the transition implies $x \leq c$ for some constant c. More precisely: for $(q, Z, Y) \xrightarrow{(q,g,R,q')} (q', Z', Y')$, the guard g implies $(x \leq c)$. A variable is *reset* by the transition if it belongs to the reset set R of the transition.

We say that a path is *blocked* if there is a variable that is bounded infinitely often and reset only finitely often by the transitions on the path. Otherwise the path is called *unblocked*.

Theorem 3. *A TBA $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition iff there exists an unblocked path in $GZG^a(\mathcal{A})$ visiting both an accepting node and a clear node infinitely often.*

The proof of Theorem 3 follows from Lemmas 5 and 6 below. We omit the proof of the first of the two lemmas and concentrate on a more difficult Lemma 6. It is here that the third component of states is used.

At the beginning of the section we had recalled that the progress criterion in [1] characterizes the paths in region graphs that have non-Zeno instantiations. We had mentioned that it cannot be directly extended to zone graphs since their transitions are not pre-stable. Lemma 6 below shows that by slightly complicating the zone graph we can recover a result very similar to Lemma 4.13 in [1].

Lemma 5. *If $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition, then in $GZG^a(\mathcal{A})$ there is an unblocked path visiting both an accepting node and a clear node infinitely often.*

Lemma 6. *Suppose $GZG^a(\mathcal{A})$ has an unblocked path visiting infinitely often both a clear node and an accepting node then $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition.*

Proof. Let σ be a path in $GZG^a(\mathcal{A})$ as required by the assumptions of the lemma (without loss of generality we assume every alternate transition is a τ transition):

$$(q_0, Z_0, Y_0) \xrightarrow{\tau} (q_0, Z_0, Y_0') \xrightarrow{t_0} \cdots (q_i, Z_i, Y_i) \xrightarrow{\tau} (q_i, Z_i, Y_i') \xrightarrow{t_i} \cdots$$

Take a corresponding path in $ZG^a(\mathcal{A})$ and one instance $\rho = (q_0, \nu_0), (q_1, \nu_1) \ldots$ that exists by Theorem 2. If it is non-Zeno then we are done.

Suppose ρ is Zeno. Let X^r be the set of variables reset infinitely often on σ. By assumption on σ, every variable not in X^r is bounded only finitely often. Since ρ is Zeno, there is an index m such that the duration of the suffix of the run starting from (q_m, ν_m) is bounded by $1/2$, and no transition in this suffix bounds a variable outside X^r. Let $n > m$ be such that every variable from X^r is reset between m and n. Observe that $\nu_n(x) < 1/2$ for every $x \in X^r$.

Take positions i, j such that $i, j > n$, $Y_i = Y_j = \emptyset$ and all the variables from X^r are reset between i and j. We look at the part of the run ρ:

$$(q_i, \nu_i) \xrightarrow{\delta_i, t_i} (q_{i+1}, \nu_{i+1}) \xrightarrow{\delta_{i+1}, t_{i+1}} \ldots (q_j, \nu_j)$$

and claim that every sequence of the form

$$(q_i, \nu_i') \xrightarrow{\delta_i, t_i} (q_{i+1}, \nu_{i+1}') \xrightarrow{\delta_{i+1}, t_{i+1}} \ldots (q_j, \nu_j')$$

is a part of a run of $\mathcal{A}$ provided there is $\zeta \in \mathbb{R}_{\geq 0}$ such that the following three conditions hold for all $k = i, \ldots, j$:

1. $\nu_k'(x) = \nu_k(x) + \zeta + 1/2$ for all $x \notin X^r$,
2. $\nu_k'(x) = \nu_k(x) + 1/2$ if $x \in X^r$ and x has not been reset between i and k.
3. $\nu_k'(x) = \nu_k(x)$ otherwise, i.e., when $x \in X^r$ and x has been reset between i and k.

Before proving this claim, let us explain how to use it to conclude the proof. The claim shows that in (q_i, ν_i) we can pass $1/2$ units of time and then construct a part of the run of $\mathcal{A}$ arriving at (q_j, ν'_j) where $\nu'_j(x) = \nu_j(x)$ for all variables in X^r, and $\nu'_j(x) = \nu_j(x) + 1/2$ for other variables. Now, we can find $l > j$, so that the pair (j, l) has the same properties as (i, j). We can pass $1/2$ units of time in j and repeat the above construction getting a longer run that has passed $1/2$ units of time twice. This way we construct a run that passes $1/2$ units of time infinitely often. By the construction it passes also infinitely often through accepting nodes.

It remains to prove the claim. Take a transition $(q_k, \nu_k) \xrightarrow{\delta_k, t_k} (q_{k+1}, \nu_{k+1})$ and show that $(q_k, \nu'_k) \xrightarrow{\delta_k, t_k} (q_{k+1}, \nu'_{k+1})$ is also a transition allowed by the automaton. Let g and R be the guard of t_k and the reset of t_k, respectively.

First we need to show that $\nu'_k + \delta_k$ satisfies the guard of t_k. For this, we need to check if for every variable $x \in X$ the constraints in g concerning x are satisfied. We have three cases:

- If $x \notin X^r$ then x is not bounded by the transition t_k, that means that in g the constraints on x are of the form $(x > c)$ or $(x \geq c)$. Since $(\nu_k + \delta_k)(x)$ satisfies these constraints so does $(\nu'_k + \delta_k)(x) \geq (\nu_k + \delta_k)(x)$.
- If $x \in X^r$ and it is reset between i and k then $\nu'_k(x) = \nu_k(x)$ so we are done.
- Otherwise, we observe that $x \notin Y_k$. This is because $Y_i = \emptyset$, and then only variables that are reset are added to Y. Since x is not reset between i and k, it cannot be in Y_k. By definition of transitions in $GZG^a(\mathcal{A})$ this means that $g \wedge (x > 0)$ is consistent. We have that $0 \leq (\nu_k + \delta_k)(x) < 1/2$ and $1/2 \leq (\nu'_k + \delta_k)(x) < 1$. So $\nu'_k + \delta_k$ satisfies all the constraints in g concerning x as $\nu_k + \delta_k$ does.

This shows that there is a transition $(q_k, \nu'_k) \xrightarrow{\delta'_k, t_k} (q_{k+1}, \nu')$ for the uniquely determined $\nu' = [R](\nu'_k + \delta_k)$. It is enough to show that $\nu' = \nu'_{k+1}$. For variables not in X^r it is clear as they are not reset. For variables that have been reset between i and k this is also clear as they have the same values in ν'_{k+1} and ν'. For the remaining variables, if a variable is not reset by the transition t_k then condition (2) holds. If it is reset then its value in ν' becomes 0; but so it is in ν'_{k+1}, and so the third condition holds. This proves the claim.

Finally, we provide an explanation as to why the proposed solution does not produce an exponential blowup. At first it may seem that we have gained nothing because when adding arbitrary sets Y we have automatically caused exponential blowup to the zone graph. We claim that this is not the case for the part of $GZG^a(\mathcal{A})$ reachable from the initial node, namely a node with the initial state of $\mathcal{A}$ and the zone putting every clock to 0.

We say that a zone *orders clocks* if for every two clocks x, y, the zone implies that at least one of $x \leq y$, or $y \leq x$ holds.

Lemma 7. *If a node with a zone Z is reachable from the initial node of the zone graph $ZG^a(\mathcal{A})$ then Z orders clocks. The same holds for $GZG^a(\mathcal{A})$.*

Suppose that Z orders clocks. We say that a set of clocks Y *respects the order given by* Z if whenever $y \in Y$ and Z implies $x \leq y$ then $x \in Y$.

Lemma 8. *If a node (q, Z, Y) is reachable from the initial node of the zone graph $GZG^a(\mathcal{A})$ then Y respects the order given by Z.*

Lemma 7 follows since a transition from a zone that orders clocks gives back a zone that orders clocks, and the $Approx_M$ operator approximates it again to a zone that orders clocks. Notice that the initial zone clearly orders clocks. The proof of Lemma 8 proceeds by an induction on the length of a path. The above two lemmas give us the desired bound.

Theorem 4. *Let $|ZG^a(\mathcal{A})|$ be the size of the zone graph, and $|X|$ be the number of clocks in $\mathcal{A}$. The number of reachable nodes of $GZG^a(\mathcal{A})$ is bounded by $|ZG^a(\mathcal{A})|.(|X|+1)$.*

The theorem follows directly from the above two lemmas. Of course, imposing that zones have ordered clocks in the definition of $GZG^a(\mathcal{A})$ we would get the same bound for the entire $GZG^a(\mathcal{A})$.

4 Algorithm

We use Theorem 3 to construct an algorithm to decide if an automaton $\mathcal{A}$ has a non-Zeno run satisfying the Büchi condition. This theorem requires to find an unblocked path in $GZG^a(\mathcal{A})$ visiting both an accepting state and a clear state infinitely often. This problem is similar to that of testing for emptiness of automata with generalized Büchi conditions as we need to satisfy two infinitary conditions at the same time. The requirement of a path being unblocked adds additional complexity to the problem. The best algorithms for testing emptiness of automata with generalized Büchi conditions are based on strongly connected components (SCC) [14,10]. So this is the way we take here.

We apply a variant of Tarjan's algorithm for detecting maximal SCCs in $GZG^a(\mathcal{A})$. During the computation of the maximal SCCs, we keep track of whether an accepting node and a clear node have been seen. For the unblocked condition we use two sets of clocks UB_Γ and R_Γ that respectively contain the clocks that are bounded and the clocks that are reset in the SCC Γ. At the end of the exploration of Γ we check if:

1. we have passed through an accepting node and a clear node,
2. there are no blocking clocks: $UB_\Gamma \subseteq R_\Gamma$.

If the two conditions are satisfied then we can conclude saying that $\mathcal{A}$ has an accepting non-Zeno run. Indeed, a path passing infinitely often through all the nodes of Γ would satisfy the conditions of Theorem 3, giving a required run of $\mathcal{A}$. If the first condition does not hold then the same theorem says that Γ does not have a witness for a non-Zeno run of $\mathcal{A}$ satisfying the Büchi condition.

The interesting case is when the first condition holds but not the second. We can then discard from Γ all the edges blocking clocks from $UB_\Gamma - R_\Gamma$, and reexamine it. If Γ without discarded edges is still an SCC then we are done. If not we restart our algorithm on Γ with the discarded edges removed. Observe

that we will not do more than $|X|$ restarts, as each time we eliminate at least one clock. If after exploring the entire graph, the algorithm has not found a subgraph satisfying the two conditions then it declares that there is no run of $\mathcal{A}$ with the desired properties. Its correctness is based on Theorem 3.

Recall that by Theorem 4 the size of $GZG^a(\mathcal{A})$ is $|ZG^a(\mathcal{A})| \cdot |X|$. The complexity of the algorithm follows from the complexity of Tarjan's algorithm and the remark about the number of recursive calls being bounded by the number of clocks. We hence obtain the following.

Theorem 5. *The above algorithm is correct and runs in time* $\mathcal{O}(|ZG^a(\mathcal{A})|.|X|^2)$.

5 Conclusions

Büchi non-emptiness problem is one of the standard problems for timed automata. Since the paper introducing the model, it has been widely accepted that addition of one auxiliary clock is an adequate method to deal with the problem of Zeno paths. This technique is also used in the recently proposed zone based algorithm for the problem [16].

In this paper, we have argued that in some cases the auxiliary clock may cause exponential blowup in the size of the zone graph. We have proposed another method that is based on a modification of the zone graph. The resulting graph grows only by a factor that is linear in the number of clocks. In our opinion, the efficiency gains of our method outweigh the fact that it requires some small modifications in the code dealing with zone graph exploration.

It is difficult to estimate how often introducing the auxiliary clock may cause an exponential blowup. The example in Figure 1 suggests that the problem appears when there is a blocked cycle containing an accepting state. A prototype implementation of our algorithm shows that a blowup occurs in the Train-Gate example (e.g. [11]) when checking for bounded response to train access requests. For 2 trains, the zone graph has 988 zones whereas after adding the auxiliary clock it blows to 227482 zones. The guessing zone graph has 3840 zones. To be fair, among the 227482 zones, 146061 are accepting with a self-loop, so in this example any on-the-fly algorithm should work rather quickly. Our prototype visits 1677 zones (in $0.42s$). The example from Figure 1 with $n = 10$ and $d = 1$ has a zone graph with 151 zones and a guessing zone graph with 1563 zones. Its size grows to 36007 when adding the extra clock. Raising d to 15, we obtain 151, 1563 and 135444 zones respectively, which confirms the expected blowup.

It is possible to apply the modification to the zone graph on-the-fly. It could also be restricted only to strongly connected components having "zero checks". This seems to be another source of big potential gains. We are currently working on an on-the-fly optimized algorithm. The first results are very encouraging. Often our prototype implementation solves the emptiness problem at the same cost as reachability when the automaton has no Zeno accepting runs. For instance, the zone graph for Fischer's protocol with 4 processes has 46129 zones and is computed in $14.22s$[1]. To answer the mutual exclusion problem it is necessary

[1] On a 2.4GHz Intel Core 2 Duo MacBook with 2GB of memory.

to visit the entire zone graph. Our algorithm does it in 15.77s. Applying the construction from [17] we get the graph with 96913 zones, and it takes 37.10s to visit all of them. Hence, even in this example, where all the runs are non-Zeno, adding one clock has a noticeable impact.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. Theoretical Computer Science 126(2), 183–235 (1994)
2. Alur, R., Madhusudan, P.: Decision problems for timed automata: A survey. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 1–24. Springer, Heidelberg (2004)
3. Behrmann, G., David, A., Larsen, K.G., Haakansson, J., Pettersson, P., Yi, W., Hendriks, M.: Uppaal 4.0. In: QEST'06, pp. 125–126 (2006)
4. Bérard, B., Bouyer, B., Petit, A.: Analysing the pgm protocol with UPPAAL. Int. Journal of Production Research 42(14), 2773–2791 (2004)
5. Berthomieu, B., Menasche, M.: An enumerative approach for analyzing time petri nets. In: IFIP Congress, pp. 41–46 (1983)
6. Bouyer, P.: Forward analysis of updatable timed automata. Formal Methods in System Design 24(3), 281–320 (2004)
7. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: Kronos: a mode-checking tool for real-time systems. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 546–550. Springer, Heidelberg (1998)
8. 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)
9. 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)
10. Gaiser, A., Schwoon, S.: Comparison of algorithms for checking emptiness on büchi automata. In: MEMICS'09, pp. 69–77 (2009)
11. Gómez, R., Bowman, H.: Efficient detection of zeno runs in timed automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 195–210. Springer, Heidelberg (2007)
12. Havelund, K., Skou, A., Larsen, K., Lund, K.: Formal modeling and analysis of an audio/video protocol: An industrial case study using UPPAAL. In: RTSS'97, pp. 2–13 (1997)
13. Jessen, J.J., Rasmussen, J.I., Larsen, K.G., David, A.: Guided controller synthesis for climate controller using UPPAAL TiGA. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 227–240. Springer, Heidelberg (2007)
14. 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)
15. Tripakis, S.: Verifying progress in timed systems. In: Katoen, J.-P. (ed.) AMAST-ARTS 1999, ARTS 1999, and AMAST-WS 1999. LNCS, vol. 1601, pp. 299–314. Springer, Heidelberg (1999)
16. Tripakis, S.: Checking timed büchi emptiness on simulation graphs. ACM Transactions on Computational Logic 10(3) (2009)
17. Tripakis, S., Yovine, S., Bouajjani, A.: Checking timed büchi automata emptiness efficiently. Formal Methods in System Design 26(3), 267–292 (2005)

MERIT: An Interpolating Model-Checker

Nicolas Caniart

LaBRI, Université de Bordeaux - CNRS UMR 5800
caniart@labri.fr

Abstract. We present the tool MERIT, a CEGAR model-checker for safety properties of counter-systems, which sits in the Lazy Abstraction with Interpolants (LAWI) framework. LAWI is parametric with respect to the interpolation technique and so is MERIT. Thanks to its open architecture, MERIT makes it possible to experiment new refinement techniques without having to re-implement the generic, technical part of the framework. In this paper, we first recall the basics of the LAWI algorithm. We then explain two heuristics in order to help termination of the CEGAR loop: the first one presents different approaches to symbolically compute interpolants. The second one focuses on how to improve the unwinding strategy. We finally report our experimental results, obtained using those heuristics, on a large amount of classical models.

1 Motivations

Lazy Abstraction with interpolants (LAWI). [8] is a generic technique to verify the safety of a system. It builds a tree by unwinding the control structure of the system. Each edge of this tree represents a transition between two control points, and each node is labeled by an over-approximation of the actual reachable configuration at that node.

LAWI follows the CEGAR [3] paradigm. It loops over three steps: *explore*, *check*, and *refine*. The *explore* step expands the reachability tree by unwinding the control structure. At first, one starts from a very coarse abstraction of the system, ignoring the transition effect, just checking the reachability of control locations marked as bad. For that reason, reaching a bad location does not necessarily mean that the system is unsafe. The *check* step looks at a branch leading to a bad location, and tries to prove that it is spurious, that is, not feasible in the actual system. If it fails, then the system is unsafe and an error trace is reported. If it succeeds, then the unwinding must be refined to eliminate this spurious path. The *refine* step consists in labeling each node on the branch by an *interpolant* that over-approximates more closely the actual configurations. This explains the term *lazy* [5]: the refinement occurs only on a branch, not on the whole tree.

Since the unwinding is in general infinite, the algorithm might not terminate. To help termination, LAWI uses a *covering* relation between nodes. Under some conditions one can guarantee that any configuration reachable from a node s is also reachable from a node t in the tree. Node t then covers node s, which prevents from having to explore the subtree rooted at s, thus limiting the tree growth. LAWI terminates when all leaves in the unwinding tree are covered. However, since nodes are relabeled during the refine step, a covered node may be uncovered, so that termination can still not be guaranteed.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 162–166, 2010.

Interpolation with Symbolic Computation. Let us explain more precisely how to refine a spurious path $s_0 \xrightarrow{\tau_1} s_1 \xrightarrow{\tau_2} \cdots \xrightarrow{\tau_n} s_n$, where each s_i is a node of the unwinding and each τ_i a transition of the system. An interpolant for this path [8] is given by sets $I_0, I_1, \ldots, I_n$ such that I_0 over-approximates the initial set of states, $\tau_k(I_k) \subseteq I_{k+1}$ for any k and $I_n = \emptyset$. Such a path interpolant witnesses that the path is spurious. In general, there are several path interpolants. The choice of the interpolant affects the algorithm behavior (and termination), since the covering relation depends on it.

In [8] theorem proving and Craig interpolation is used to compute interpolants. Our work focuses on model-checking counter-systems [9] with unbounded variables. The transition relations are encoded in the Presburger logic. So far, Craig interpolation algorithms are known only for fragments of this logic [6]. In [4] it is showed how to compute interpolants symbolically. We have chosen to first experiment symbolic computation techniques in our model-checker, using the TaPAS tools [7]: an interface to various automata- or formula-based symbolic set representations.

2 The MERIT Model-Checker

MERIT is a model-checker for counter-systems based on the LAWI framework. We discuss its architecture, and present two improvements to the generic algorithm that we both implemented in MERIT.

Open architecture. An interesting feature of the LAWI algorithm is that it offers a clear split between operations that work only on the control graph of the transition system, and those that compute interpolants. Thanks to this, we were able to use virtually any kind of techniques to compute interpolants: theorem proving, SMT-solvers, or symbolic computation. All operations or queries made on interpolants are implemented behind the interface of a single module, called a *refinery*. Changing the way interpolants are computed is just a matter of changing the refinery. We currently have one fully functional refinery based on the TAPAS framework [7] and we are working on an SMT-solver based refinery.

Optimizing the interpolants. MERIT implements the classical symbolic weakest pre- and strongest post-conditions computations, which provide path interpolants [4]. They are named weakest (resp. strongest) since they are the maximal (resp. minimal) sets of configurations that can appear on a path interpolant. MERIT also implements two original, and in practice more efficient techniques, that we both experimented.

- The first uses post- symbolic computation to find the closest node from the root of the refined branch where the reachability set becomes empty. From that node, it replays the trace backwards, computing the weakest path interpolant. This way, we obtain shorter branches than when using a weak-pre computation, but weaker interpolants on the higher part of the branch than when using a post computation. The idea of the heuristic is to make it easier to cover nodes by producing large interpolants high in the tree.
- The second one is the dual of the first: it starts with a backward symbolic computation, and it then tightens interpolants on the lower part of the branch using strongest interpolants.

Experimental results for the later technique, called *cut-post*, are given in Table 1.

Tuning the unwinding strategy: BFS vs. DFS. Our experiments also show that the strategy used to expand the tree has an impact on the algorithm termination. In [8] it is suggested to use a DFS strategy to expand the tree. Indeed a DFS strategy seems more adequate than a BFS one: suppose that the algorithem has to visit nodes at depth d. With systems having control locations of out-degree k, it is then necessary to compute and store at least k^d nodes. In practice models with few control locations and high out-degree emerge naturally[1].

The problem comes from the fact that a naive implementation of the DFS expansion strategy can behaves like a BFS. Indeed, in [8] a node is expanded by adding all its children at once: e.g. a node t gets 3 children u, v, w. Because those nodes have not been refined, they are labeled by the full variable domain. Thus each of them can cover any node added after itself (provided they correspond to the same control location). The DFS proceeds by expanding u. Suppose now that children of u are all covered by either v or w. The DFS is therefore stopped and we have to explore a new branch (in v subtree). Again, the children of v may become covered by w. We see how a "BFS-like" behavior arises. This phenomenon does occur in practice and a combinatorial blowup indeed impairs the algorithm termination.

To fix this problem, we add only one child, and pursue the DFS with it. When that child is popped from the DFS stack, we add its sibling and repeat the same process. This way we add less nodes labeled by the full variable domain, which prevents from covering uselessly. The termination condition becomes more complicated. We also have to check that we did not forget to fully expand all internal nodes. Nonetheless our experiments show that, using this strategy, our tool can cope with models where the original strategy fails. The impact on the tool performance can be drastic, as showed in Table 1.

3 Experimental Results

MERIT has been tested with a pool of about 50 infinite-state systems, ranging from broadcast protocols to programs with dynamic data-structures. The benchmark suite we use is available on the tool webpage (cf. Availability section, p. 165). The verification was successful in about 80% of the tests and MERIT detected 100% of the unsafe models.

The use of the "append one child at a time" unwinding strategy and the cut-pre or cut-post refinement techniques presented in Sec. 2 allowed MERIT to almost double the number of models it can tackle. Table 1 presents the results obtained (1) with the original algorithm, the weakest pre-conditions refinement (column Original Pre), the one child at a time algorithm with the same refinement technique (column 1 child Pre), as well as the one child algorithm with the cut-post refinement method (column 1 child cut-post). This shows how much we improved from the original algorithm. We also compare our tool to the tools FAST[2] and ASPIC[3] because they make use of acceleration [1] techniques which are particularly efficient on the models we use to test MERIT. However MERIT is more efficient than FAST and ASPIC on many models.

[1] Like for distributed system models, where the global control structure encoded by variables.

[2] Available at `http://www.lsv.ens-cachan.fr/Software/fast/`

[3] Available at `http://laure.gonnord.org/pro/aspic/aspic.html`

Table 1. Benchmark results

				Original Pre			1 child Pre			1 child cut-post			FAST	ASPIC
MODEL	V	T	O	N	R	TIME	N	R	TIME	N	R	TIME	TIME	TIME
ILLINOIS	5	9	S	4152	415	2.12	777	388	1.72	-	-	TOUT	1.75	?
insert	48	51	S	74	11	1.28	70	11	1.35	70	11	1.25	3.97	0.14
MESI	5	4	S	287	57	1.42	107	53	1.05	35	17	1.13	1.71	?
merge	847	1347	S	6661	944	27.77	5413	952	40.34	189	30	3.79	TOUT	2.27
MOESI	6	4	S	27	5	1.23	11	5	1.14	35	17	1.16	1.36	?
train	7	12	S	20878	4302	268.64	205	101	1.51	1531	765	13.55	2.29	?
deleteAll	18	19	U	13	2	1.10	13	2	0.98	13	2	1.18	1.0	0.11

Legend: V = # of variables; T: # of transitions; O: outcome, S means safe, U unsafe, ? tool does not know ; N: # tree nodes, R: # refinements. TOUT means time-out, MOUT memory outage.

4 Conclusion and Development Perspectives

In this paper we presented MERIT, a model-checker tool that use symbolic interpolant computation techniques. It implements the Lazy Abstraction with Interpolants algorithm [8]. The models we experimented are particularly suited for acceleration techniques. However MERIT was able to tackle many models without using acceleration.

Short-term goals: One of our short term goals is to get a fully functionnal SMT-Solver based refinery, to see how such a technique can compete with symbolic ones.

Mid-term goals: We noticed that some refinement techniques are complementary: they succeed on different sets of models and the union of those sets almost covers the whole set of models. We tried hybrid refinement techniques that combine them. This allowed MERIT tackle more models. However the problem of choosing, on the fly, the proper interpolation technique for a branch is still an open problem.

Finally, our experiments showed that some difficult examples would benefit from *acceleration* [2] techniques like the `train` model in Tab. 1. However combining LAWI and acceleration is still an open question. Acceleration is costly and the trade-off between that cost and the benefit for the cover relation has to be investigated.

Availability. MERIT is available under free software license at `http://www.labri.fr/~caniart/merit.html`.

References

[1] Boigelot, B.: On Iterating Linear Transformations Over Recognizable Sets of Integers. Theoretical Computer Science 309(1-3), 413–468 (2003)

[2] Caniart, N., Fleury, E., Leroux, J., Zeitoun, M.: Accelerating interpolation-based model-checking. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 428–443. Springer, Heidelberg (2008)

[3] 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)

[4] Esparza, J., Kiefer, S., Schwoon, S.: Abstraction refinement with Craig interpolation and symbolic pushdown systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 489–503. Springer, Heidelberg (2006)
[5] Henzinger, T.A., Jhala, R., Majumbar, R., Sutre, G.: Lazy Abstraction. In: Proc. of POPL'02, pp. 58–70 (2002)
[6] Jain, H., Clarke, E.M., Grumberg, O.: Efficient Craig interpolation for linear diophantine (dis)equations and linear modular equations. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 254–267. Springer, Heidelberg (2008)
[7] Leroux, J., Point, G.: Tapas: The Talence Presburger Arithmetic Suite. In: Proc. of TACAS'09. LNCS, vol. 5505, pp. 182–185. Springer, Heidelberg (2009)
[8] 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)
[9] Minsky, M.L.: Computation: Finite and Infinite Machines, June 1967. Prentice-Hall, Englewood Cliffs (June 1967)

Breach, A Toolbox for Verification and Parameter Synthesis of Hybrid Systems

Alexandre Donzé

Verimag Laboratory 2, Avenue de Vignates, 38610 Gières France
donze@imag.fr
http://www-verimag.imag.fr/~donze/tool.shtml

Abstract. We describe `Breach`, a Matlab/C++ toolbox providing a coherent set of simulation-based techniques aimed at the analysis of deterministic models of hybrid dynamical systems. The primary feature of `Breach` is to facilitate the computation and the property investigation of large sets of trajectories. It relies on an efficient numerical solver of ordinary differential equations that can also provide information about sensitivity with respect to parameters variation. The latter is used to perform approximate reachability analysis and parameter synthesis. A major novel feature is the robust monitoring of metric interval temporal logic (MITL) formulas. The application domain of `Breach` ranges from embedded systems design to the analysis of complex non-linear models from systems biology.

1 Introduction

Model-based analysis and design techniques for complex systems with parameters uncertainty rely mostly on extensive simulation. Hybrid systems feature a mix of continuous and discrete components and most often their number of possible behaviors is infinite, rendering formal design by exhaustive simulation impossible. Instead, reachability analysis is used to generate over-approximations of the set of possible behaviors to prove that they all satisfy a given property. Efficient techniques and tools exist for hybrid systems with linear continuous dynamics [ADF+06] but to the best of our knowledge, no tool can be readily scalable for hybrid non-linear dynamics, as can be, for instance, simulation. Hence the original idea (also following [KKMS03, GP]) that lead to the development of `Breach` was to estimate dense sets reachable by the system based only on a finite (though possibly large) number of simulations. `Breach` implements this idea and was used, e.g., to produce the results presented in [DKR, DCL09]. It has now matured into a more general exploration tool for hybrid dynamical systems with uncertain parameters, with a convenient graphical user interface and the possibility to write MITL formulas and efficiently monitor their satisfaction robustness.

2 Hybrid Systems Definition and Simulation

`Breach` deals with piecewise-continuous hybrid systems specified as

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 167–170, 2010.

$$\begin{cases} \dot{\mathbf{x}} = f(q, \mathbf{x}, \mathbf{p}),\ \mathbf{x}(0) = \mathbf{x}_0 \\ q^+ = e(q^-, \lambda),\ q(0) = q_0 \\ \lambda = \text{sign}(g(\mathbf{x})) \end{cases} \quad (1)$$

where $\mathbf{x} \in \mathbb{R}^n$ is the state, $\mathbf{p} \in \mathcal{P} \subset \mathbb{R}^{n_p}$ is the *parameter vector*, $q \in \mathcal{Q}$ is the discrete state, g is the guard function mapping $\mathbb{R}^n$ to $\mathbb{R}^{n_g}$, and sign is the usual sign function extended to vectors. The function e is the *event function* which updates the discrete state when the sign of one component of g changes. A trajectory $\xi_{\mathbf{p}}$ is a function from $\mathbb{T} = \mathbb{R}^+$ to $\mathbb{R}^n$ satisfying (1) for all t in $\mathbb{T}$. For convenience, the initial state $\mathbf{x}_0$ is included in the parameter vector $\mathbf{p}$. Thus $\xi_{\mathbf{p}}(0) = \mathbf{x}_0 = (x_{0,1}, x_{0,2}, \ldots, x_{0,n})$ where for all $i \leq n$, $x_{0,i} = p_i$. `Breach` implements a standard discontinuity locking method for the simulation of such systems, based on CVodes ODE solver[1]. *Sensitivity analysis* consists in measuring the influence of a parameter change $\delta\mathbf{p}$ on a trajectory $\xi_{\mathbf{p}}$. A first-order approximation can be obtained by the Taylor expansion

$$\xi_{\mathbf{p}+\delta\mathbf{p}}(t) = \xi_{\mathbf{p}}(t) + \frac{\partial \xi_{\mathbf{p}}}{\partial \mathbf{p}}(t)\ \delta\mathbf{p} + \varphi(t, \delta\mathbf{p}) \text{ where } \varphi(t, \delta\mathbf{p}) = \mathcal{O}\big(\|\delta\mathbf{p}\|^2\big) \quad (2)$$

The derivative of $\xi_p(t)$ with respect to $\mathbf{p}$ in the right hand side of (2) is called the *sensitivity matrix* and denoted as $S_{\mathbf{p}}(t) = \frac{\partial \xi_{\mathbf{p}}}{\partial \mathbf{p}}(t)$. CVodes implements a common method to compute it for an ODE, by integrating a linear time varying ODE satisfied by $S_{\mathbf{p}}(t)$. For hybrid systems such as (1), the sensitivity equation can be solved between two consecutive events but S_p is discontinuous when a guard is crossed. `Breach` implements the computation of the discontinuity jumps provided that the guard functions are smooth enough at the crossing point (see [DKR] for more details).

3 Main Features Overview

Reachability using sensitivity analysis. The reachable set induced by a set of parameters $\mathcal{P}$ at time t is $\mathcal{R}_t(\mathcal{P}) = \bigcup_{\mathbf{p}\in\mathcal{P}} \xi_{\mathbf{p}}(t)$. We showed in [DKR] that it can be approximated by using sensitivity analysis. Let $\mathbf{p}$ and $\mathbf{p}'$ be two parameter vectors in $\mathcal{P}$ and assume that we computed the trajectory $\xi_{\mathbf{p}}$ and the sensitivity matrix $S_{\mathbf{p}}$ at time t. Then we can use $\xi_{\mathbf{p}}(t)$ and $S_{\mathbf{p}}(t)$ to estimate $\xi_{\mathbf{p}'}(t)$. We denote this estimate by $\hat{\xi}^{\mathbf{p}}_{\mathbf{p}'}(t)$. The idea is to drop higher order terms in the Taylor expansion (2), which gives $\hat{\xi}^{\mathbf{p}}_{\mathbf{p}'}(t) = \xi_{\mathbf{p}}(t) + S_{\mathbf{p}}(t)(\mathbf{p}' - \mathbf{p})$. If we extend this estimate to all parameters $\mathbf{p}'$ in $\mathcal{P}$, we get the following estimate for the reachable set $\mathcal{R}_t(\mathcal{P})$: $\hat{\mathcal{R}}^{\mathbf{p}}_t(\mathcal{P}) = \bigcup_{\mathbf{p}'\in\mathcal{P}} \hat{\xi}_{\mathbf{p}'}(t) = \{\xi_{\mathbf{p}} - S_{\mathbf{p}}(t)\mathbf{p}\} \oplus S_{\mathbf{p}}(t)\mathcal{P}$. If the approximation with one trajectory is too coarse, it can be improved by refining $\mathcal{P}$ into smaller sets until an error tolerance factor is satisfied [DKR]. The process is illustrated in Fig. 1. It converges quadratically, although the error cannot be formally bounded in general (as is the case with numerical simulation). In [DKR], a local iterative refinement of the parameters boxes for which the reachable set intersects a bad set is used to synthesize sets of trajectories satisfying a safety property.

[1] See `https://computation.llnl.gov/casc/sundials/main.html`

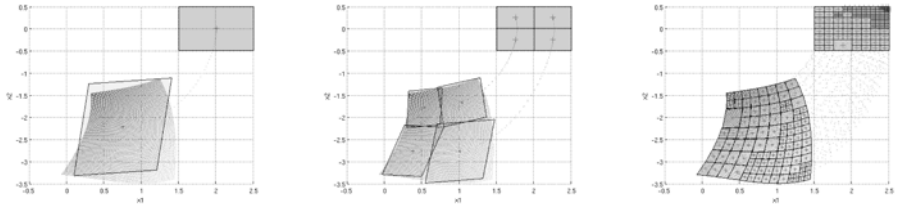


Fig. 1. Approximation of the reachable set for a Van der Pol equation using one trajectory, four trajectories, and an automatic refinement produced by the reachability routine of `Breach` with control of the error

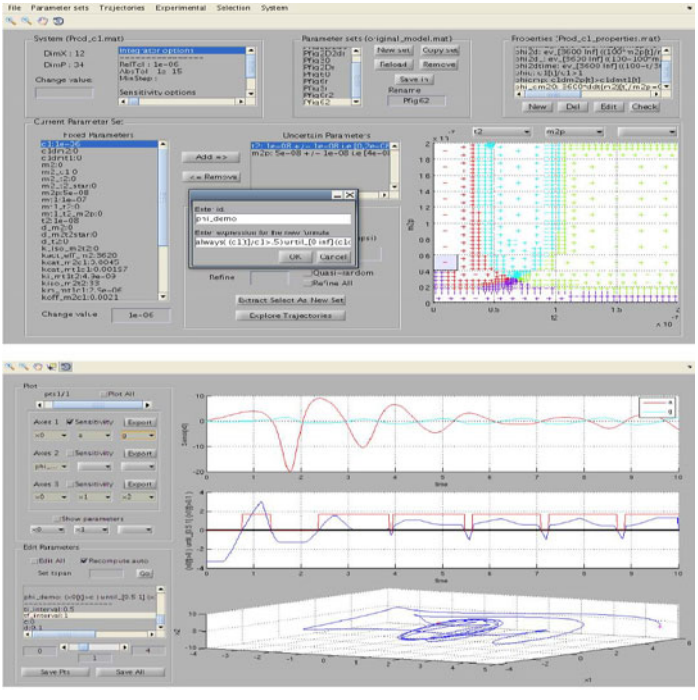


Fig. 2. Top: main window of `Breach`, manipulating parameter sets and properties. Each cross in the plot is for a given parameter vector and all parameters with the same color satisfy the same set of properties. Bottom: Trajectories explorer window. Each plot can display either the evolution of the state variables, their sensitivities or the robust satisfaction of MITL formulas. All parameters can be modified and the trajectories recomputed on-the-fly.

Property-driven parameters synthesis. Recently, we implemented an extension to support formulas of Signal Temporal Logic (STL), an analog extension of the Metric Interval Temporal Logic (MITL) which has the following core grammar:

$$\varphi := p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi_1 \mathcal{U}_I \varphi \tag{3}$$

The specificity of STL lies in the nature of its atomic predicates p, which are expressions of the form $y(x_0, x_1, \ldots, x_n, t) > 0$. A parser allows to specify

textually formulas with a richer grammar. E.g., a valid expression is `(x0[t]>0)=> eventually_[1,2] ((x1[t]-1>0) until_[1,2](-x2[t]-.1>0))`. `Breach` is able to monitor the Boolean satisfaction as well as the quantitative satisfaction. For p, the former is given by the sign of y (similarly to the tool AMT[2]) and the latter is given by $|y|$. For a compound formula the semantics follows that of [FP07]. We are not aware of other tools implementing this feature, except for TaLiRo[3] which `Breach` outperformed in our experiments. In particular, `Breach` does not suffer from the memory explosion problem reported in the user guide of Taliro for larger formulas. The computational time is experimentally linear in the size of the trace and the size of the formula, which is a light overhead to the cost of the simulation (see `Breach` website for examples and data). The use of sensitivity analysis for properties other than safety is still a work in progress, but `Breach` provides heuristics to find separations between parameter regions satisfying different properties. The GUI is shown on Fig. 2. In addition to defining systems and interfacing the above mentioned methods, it allows to explore the behaviors and monitor their properties by tuning the parameters on-the-fly.

4 Discussion and Future Work

`Breach` is still in very active development. For the moment, the GUI allows to specify models as general nonlinear ODES. For hybrid systems, the user still has to provide write small portions of C code to implement transitions (an example is given on the web site). Also, although in [DKR] we demonstrated the feasibility of analysing Simulink models with `Breach` , this feature still need some work in particular when the system is hybrid.

References

[ADF+06] Asarin, E., Dang, T., Frehse, G., Girard, A., Le Guernic, C., Maler, O.: Recent progress in continuous and hybrid reachability analysis (2006)

[DCL09] Donzé, A., Clermont, G., Langmead, C.: Parameter synthesis for nonlinear dynamical systems: Application to systems biology. Journal of Computational Biology 410(42) (2009)

[DKR] Donzé, A., Krogh, B., Rajhans, A.: Parameter synthesis for hybrid systems with an application to simulink models. In: Majumdar, R., Tabuada, P. (eds.) HSCC 2009. LNCS, vol. 5469, pp. 165–179. Springer, Heidelberg (2009)

[FP07] Fainekos, G., Pappas, G.: Robust sampling for mitl specifications. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 147–162. Springer, Heidelberg (2007)

[GP] Girard, A., Pappas, G.: Verification Using Simulation. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 272–286. Springer, Heidelberg (2006)

[KKMS03] Kapinski, J., Krogh, B.H., Maler, O., Stursberg, O.: On systematic simulation of open continuous systems. In: Maler, O., Pnueli, A. (eds.) HSCC 2003. LNCS, vol. 2623, pp. 283–297. Springer, Heidelberg (2003)

[2] See `http://www-verimag.imag.fr/DIST-TOOLS/TEMPO/AMT/content.html`

[3] See `http://www.public.asu.edu/~gfaineko/taliro.html`

JTLV: A Framework for Developing Verification Algorithms*

Amir Pnueli, Yaniv Sa'ar[1], and Lenore D. Zuck[2]

[1] Weizmann Institute of Science
yaniv.saar@weizmann.ac.il
[2] University of Illinois at Chicago
lenore@cs.uic.edu

1 Introduction

JTLV[1] is a computer-aided verification scripting environment offering state-of-the-art Integrated Developer Environment for algorithmic verification applications. JTLV may be viewed as a new, and much enhanced TLV [18], with Java rather than TLV-basic as the scripting language. JTLV attaches its internal parsers as an Eclipse editor, and facilitates a rich, common, and abstract verification developer environment that is implemented as an Eclipse plugin.

JTLV allows for easy access to various low-level BDD packages with a high-level Java programming environment, without the need to alter, or even access, the implementation of the underlying BDD packages. It allows for the manipulation and on-the-fly creation of BDD structures originating from various BDD packages, whether existing ones or user-defined ones. In fact, the developer can instantiate several BDD managers, and alternate between them during run-time of a single application so to gain their combined benefits.

Through the high-level API the developer can load into the Java code several SMV-like *modules* representing programs and specification files, and directly access their components. The developer can also procedurally construct such modules and specifications, which enables loading various data structures (e.g., statecharts, LSCs, and automata) and compile them into modules.

JTLV offers users the advantages of the numerous tools developed by Java's ecosystem (e.g., debuggers, performance tools, etc.). Moreover, JTLV developers are able to introduce software methodologies such as multi-threading, object oriented design for verification algorithms, and reuse of implementations.

2 JTLV: Architecture

JTLV, described in Fig. 1, is composed of three main components, the API, the Eclipse Interface, and the Core.

* This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant awarded to David Harel from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013). This material is based on work supported by the National Science Foundation, while Lenore Zuck was working at the Foundation. Any opinion, finding, and conclusions or recommendations expressed in this article are those of the author and do not necessarily reflect the views of the National Science Foundation.

[1] JTLV homepage: http://jtlv.ysaar.net

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 171–174, 2010.

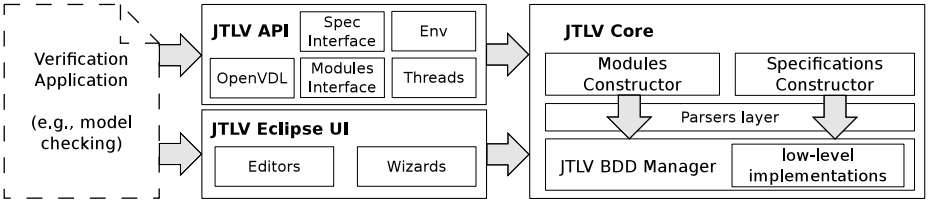


Fig. 1. JTLV Architecture

API. In the following we present a set of *sample* functionalities. An exhaustive list of the API is in [20]

- TLV-like ability to procedurally create and manipulate BDD's on-the-fly, a useful feature when dealing with abstractions and refinements ([1]);
- seamlessly alternate BDD packages at run-time (due to the factory design pattern [22]);
- save (and load) BDDs to (and from) the file system;
- load modules written in *NuSMV*-like language enriched with *Cadence SMV*-like parsing of loops and arrays of processes;
- procedurally access module's fields as well as its BDD's;
- perform basic functionalities on a module, e.g., compute successors or predecessors, feasible states, shortest paths from one state to another, etc.;
- procedurally create new modules and add, remove, and manipulate fields ;
- load temporal logic specification files;
- procedurally create and access the specification objects.

JTLV supports *thread*s, that are Java native threads coupled with dedicated BDD memory managers. Each thread can execute freely, without dependencies or synchronization with other threads. To allow for BDD-communication among threads, JTLV provides a low-level procedure that copies BDDs from one BDD manager into another. Our experience has shown that for applications that accommodate compositionality, an execution using threads outperforms its sequential counterparts.

Assisted by the API, the user can implement numerous types of verification algorithms, some mentioned in the next section. It also contains the *OpenVDL* (Open Verification Developer Library), which is a collection of known implementations enabling their reuse.

Eclipse User Interface. Porting the necessary infrastructure into Java enables plugging JTLV into Eclipse, which in turn facilitates rich new editors to module and specification languages (see website for snapshots). A new JTLV project automatically plugs-in all libraries. JTLV project introduces new builders that take advantage of the underlying parsers, and connects them to these designated new editors.

Core. The core component encapsulates the BDD implementation and parses the modules and specifications. Through the JAVA-BDD ([22]) factory design pattern, a developer can use a variety of BDD packages (e.g., CUDD [21], BUDDY [14], and CAL [19]), or design a new one. This also allows for the development of an application regardless of the BDD package used. In fact, the developer can alternate BDD packages during

run-time of a single application. The encapsulation of the memory management system allows JTLV to easily instantiate numerous BDD managers so to gain the combined benefits of several BDD packages simultaneously. This is enabled by APIs that allow for translations among BDD's generated by different packages, so that one can apply the functionality of a BDD-package on a BDD generated by another package.

3 Conclusion, Related, and Future Work

We introduced JTLV, a scripting *environment* for developing algorithmic verification applications. JTLV is not a dedicated model checker (e.g. [2,13,10]) – its goal is to provide for a convenient development environment, and thus cannot be compared to a particular model checkers. Yet, as shown in Table 1, our implementation of invariance checking at times outperforms similar computations in such model checkers.

Table 1. Performance results (in sec.) of JTLV, compared to other model checkers

Check Invariant	Muxsem 56	Bakery 7	Szymanski 6
JTLV	11	39.9	34.4
TLV	21.4	36.2	19
NuSMV	18.1	37.8	19.4
Cadence SMV	24.6	53.6	36.7

We are happy to report that JTLV already has a small, and avid, user community, including researchers from Imperial College London [15], New York University [5,4,6], Bell Labs Alcatel-Lucent [5,4,6], Weizmann Institute [8,9], Microsoft Research Cambridge, RWTH-Aachen, California Institute of Technology [24,23], GRASP Laboratory University of Pennsylvania [7], and University of California Los Angeles. In these works JTLV is applied to: Streett and Rabin Games; Synthesis of GR(k) specifications; Compositional multi-theaded model checking; Compositional LTL model checking; Verifying heap properties; Automata representation of LSCs and Statecharts; Synthesis of LSCs and of hybrid controllers.

The JTLV library (see [20]) includes numerous model checking applications, including LTL and CTL* model checking [12], fair-simulation [11], a synthesis algorithm [17], Streett and Rabin games [16], compositional model checking ([3]), and compositional multi threaded model checking [6]. The API can also facilitate the reduction of other models into the verification framework (see, e.g., [8] where LSCs are reduced to automata).

We are currently developing a new thread-safe BDD package to allow concurrent access from multiple clients. Integrating a thread-safe BDD package into JTLV will entail a new methodology, which will streamline the development of multi-threaded symbolic algorithms. This calls for an in-depth overview of many symbolic applications. We are also in the process of developing new interfaces to non-BDD managers

References

1. Balaban, I., Fang, Y., Pnueli, A., Zuck, L.D.: IIV: An Invisible Invariant Verifier. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 408–412. Springer, Heidelberg (2005)

2. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: A new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
3. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
4. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: A dash of fairness for compositional reasoning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 543–557. Springer, Heidelberg (2010)
5. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: Split: A compositional LTL verifier. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 558–561. Springer, Heidelberg (2010)
6. Cohen, A., Namjoshi, K.S., Sa'ar, Y., Zuck, L.D., Kisyova, K.I.: Parallelizing a symbolic compositional model-checking algorithm (in preparation) (2010)
7. Gazit, H.K., Ayanian, N., Pappas, G., Kumar, V.: Recycling controllers. In: IEEE Conference on Automation Science and Engineering, Washington (August 2008)
8. Harel, D., Maoz, S., Segall, I.: Using automata representations of LSCs for smart play-out and synthesis (in preparation) (2010)
9. Harel, D., Segall, I.: Synthesis from live sequence chart specifications (in preparation) (2010)
10. Holzmann, G.: Spin model checker, the: primer and reference manual. Addison-Wesley Professional, Reading (2003)
11. Kesten, Y., Piterman, N., Pnueli, A.: Bridging the gap between fair simulation and trace inclusion. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 381–392. Springer, Heidelberg (2003)
12. Kesten, Y., Pnueli, A., Raviv, L., Shahar, E.: LTL model checking with strong fairness. Formal Methods in System Design (2002)
13. Cadence Berkeley Lab. Cadence SMV (1998), http://www-cad.eecs.berkeley.edu/kenmcmil/smv
14. Nielson, J.L.: Buddy, http://buddy.sourceforge.net
15. Piterman, N.: Suggested projects (2009), http://www.doc.ic.ac.uk/~npiterma/projects.html
16. Piterman, N., Pnueli, A.: Faster solutions of rabin and streett games. In: LICS, pp. 275–284 (2006)
17. 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 (2005)
18. Pnueli, A., Shahar, E.: A platform for combining deductive with algorithmic verification. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 184–195. Springer, Heidelberg (1996)
19. Ranjan, R.K., Sanghavi, J.V., Brayton, R.K., Vincentelli, A.S.: High performance BDD package based on exploiting memory hierarchy. In: DAC'96, June 1996, pp. 635–640 (1996)
20. Sa'ar, Y.: JTLV – web API, http://jtlv.ysaar.net/resources/javaDoc/API1.3.2/
21. Somenzi, F.: CUDD: CU Decision Diagram package (1998), http://vlsi.colorado.edu/~fabio/CUDD/
22. Whaley, J.: JavaBDD, http://javabdd.sourceforge.net
23. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Automatic synthesis of robust embedded control software. submitted to AAAI'10 (2010)
24. Wongpiromsarn, T., Topcu, U., Murray, R.M.: Receding horizon control for temporal logic specifications. In: HSCC'10 (2010)

Petruchio: From Dynamic Networks to Nets

Roland Meyer[1] and Tim Strazny[2]

[1] LIAFA & CNRS
[2] University of Oldenburg

Abstract. We introduce Petruchio, a tool for computing Petri net translations of dynamic networks. To cater for unbounded architectures beyond the capabilities of existing implementations, the principle fixed-point engine runs interleaved with coverability queries. We discuss algorithmic enhancements and provide experimental evidence that Petruchio copes with models of reasonable size.

1 Introduction

Petruchio computes Petri net representations of dynamic networks, as they are the basis to automatic-verification efforts [19]. As opposed to static networks where the topology is fixed, in dynamic networks the number of components as well as connections changes at runtime. Whereas earlier tools covered only finite state models [6,23,9], Petruchio features the unbounded interconnection topologies needed when tackling software. Theoretically, the implementation rests upon recent insights on the relationship between dynamic networks and Petri nets [15,14]. Practically, the heart of our algorithm is an unconventional fixed-point computation interleaved with coverability queries.

Run on a series of benchmarks, we routinely translate systems of two hundred lines of π-calculus code into Petri nets of around 1k places within seconds. The computability threshold lies around 90k transitions, which is in turn beyond the capabilities of latest net verification tools [13]. A concurrency bug found in an automated manufacturing system and automatic verification of the gsm benchmark underline the practicability of our tool [16].

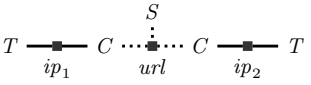
Related Work. There has been recent interest in translation-based network verification [4,3,16], Petruchio puts these efforts into practice. Besides, the well-structured transition system framework [2,5,8,1,24] as well as abstraction-based verification techniques [21,20,11,22] have been applied.

2 Foundations behind Petruchio

Online banking services are typical dynamic networks where failures have severe consequences and thus verification is required. We model this example in the π-calculus and for simplicity explain the implementation of the Petri net translation from [14]. Based on similar algorithmic ideas, the fixed-point engine in Petruchio also handles the more involved translations from [3,15].

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 175–179, 2010.

$$S(url) = url(y).(\overline{y}\langle bal\rangle \mid S(url))$$
$$C(url) = \nu ip.\overline{url}\langle ip\rangle.ip(dat).C(url)$$



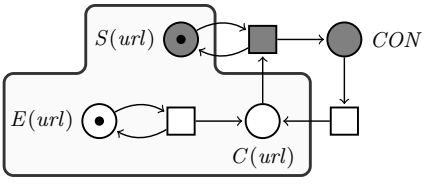


Fig. 1. π-calculus model *Bnk* of an online banking service (top left) and a reachable state represented as interconnection topology (bottom left). The structural semantics $\mathcal{N}_S[\![Bnk]\!]$ is depicted to the right, *CON* abbreviates $\nu ip.(ip(dat).C(url) \mid \overline{ip}\langle bal\rangle)$.

The overall functionality of the banking system *Bnk* is a login of the client, which spawns a new thread that displays the account balance. We detail the π-calculus model in Figure 1. The bank server $S(url)$ is located at some url and ready to receive the ip-address of a customer, $url(y)$. Upon reception, a new thread is spawned (parallel composition $\mid$). It transmits the balance, $\overline{y}\langle bal\rangle$, and terminates. The server itself is immediately ready for new requests. To guarantee proper interaction, the client sends its private (indicated by a νip quantifier) ip-address $\overline{url}\langle ip\rangle$ and waits on this channel for data. We assume an environment $E(url)$ that generates further customers.

Translation. Although the banking service exhibits an unbounded number of connection topologies, there exists a finite basis of connection *fragments* they are built from. Fragments are maximal subgraphs induced by private channels and can be determined in linear time by minimising the scopes of the quantifiers. For instance, a private connection between client and thread is fragment $\nu ip.(ip(dat).C(url) \mid \overline{ip}\langle bal\rangle)$. It is present twice in the example state in Figure 1.

For verification purposes, the *structural semantics* translates dynamic networks into Petri nets. Every reachable fragment yields a place, communications inside and between fragments determine the transitions, and the initial state is the decomposition of the system's initial state into fragments. The running example is represented by the Petri net $\mathcal{N}_S[\![Bnk]\!]$ in Figure 1.

An *isomorphism* between the transition systems, $Sys =_{iso} \mathcal{N}_S[\![Sys]\!]$, proves the net representation suitable for model checking purposes. In fact, it is a lower bound on the information required for verifying topological properties. This follows from a *full abstraction* result wrt. syntactic equivalence, $Sys \equiv Sys'$ iff $\mathcal{N}_S[\![Sys]\!] = \mathcal{N}_S[\![Sys']\!]$, and the descriptive power of topological logics [7].

3 Algorithmic Aspects

The declarative definition of the structural semantics leaves the problem of its computability open. Taking a classical view from denotational semantics, we understand it as an unconventional least fixed-point on a particular set of nets. A dynamic network *Sys* gives rise to a function ϕ_{Sys} on nets. As an example, consider the subnet N shown in the box in Figure 1. An application $\phi_{Bnk}(N)$ extends

it by the communication between client and server (dark). The least fixed-point of such a ϕ_{Sys} is in fact the structural semantics, $\mathcal{N}_{\mathcal{S}}[\![Sys]\!] = lfp(\phi_{Sys})$. Thanks to continuity, we can compute it by iterating the function on the empty net, $lfp(\phi_{Sys}) = \sqcup\{\phi^n_{Sys}(\mathcal{N}_\emptyset) \mid n \in \mathbb{N}\}$. The algorithm terminates precisely on systems with a finite structural semantics. They are *completely characterised* by the existence of a finite basis of fragments [14].

Leading yardstick to a practical implementation is the efficient *computation of extensions* and the quick *insertion of places*.

Computing Extensions. An application of function ϕ_{Sys} determines the set of transitions the net has to be extended with. Transitions between fragments rely on pairs (F, G) of *potential communication partners*. Hashing the leading communication channels, they can be determined in constant time. Each such pair then needs a *semantic confirmation* of F and Gs simultaneous reachability. We reduce it to a coverability problem in the Petri net built so far and implement strategies to *avoid* unnecessary queries and *speed-up* coverability checks.

To reduce the number of checks, PETRUCHIO augments the breadth-first fixed-point computation with dedicated depth-first searches. Whenever fragments F and G are found simultaneously markable, we build their *internal closure* $cl(F)$. It consists of all fragments reachable from F with internal communications. By definition, containment in the internal closure is a semantic confirmation for all potential communication partners $F' \in cl(F)$ and $G' \in cl(G)$. Their transitions can be added without further coverability queries.

Despite the advantage of incremental computability [12], Karp and Miller graphs turned out impractical for coverability checks due to their size. Instead, we perform independent backwards searches [2] that we prune with knowledge about place bounds. These bounds are derived from place invariants, and we currently use an incomplete cubic time algorithm. Our experiments show that already non-optimal bounds dramatically speed-up the backwards search.

Inserting Places. Every newly discovered fragment F in $\phi_{Sys}(N)$ has to be compared for syntactic equivalence $\equiv$ with the places in the original net N. Since these checks $F \equiv G$ are graph isomorphism complete [10], we implemented a technique in PETRUCHIO to minimise their number.

We abstract fragments to so-called *signatures* $sig(F)$. As equality of these signatures is necessary for syntactic equivalence, they allow us to quickly refute non-equivalent pairs $F \not\equiv G$. Technically, the theory rests upon functions α that are invariant under syntactic equivalence, $F \equiv G$ implies $\alpha(F) = \alpha(G)$. A signature is a combination of these *indicator values*, $sig(F) := \alpha(F).\beta(F)\ldots$ We use ten values, ranging from number of free names to sequences of input and output actions. All of them are computable in linear time.

As all indicator values stem from totally ordered domains, the lexicographic order on signatures is total. When a new fragment is inserted, we can thus rely on a (logarithmic) binary search for candidates $sig(F) = sig(G)$ that need to be checked for syntactic equivalence. The check itself is implemented in PETRUCHIO and we provide the option to hand over larger instances to a *graph isomorphism solver* that we integrated in black-box fashion [10,17].

Experimental Evaluation. The implementation encapsulates coverability checker and fixed-point engine into separate threads that run loosely coupled. We demonstrate its efficiency on the gsm handover procedure [18] and an automatic manufacturing system [16]. Note that HTS_P with a parametric number of transport vehicles yields a smaller net than the concrete model HTS_C with six of them, underpinning the need for unbounded verification techniques. For each model, we give loc, Petri net size (places, transitions, edges), and compile-time on an AMD Athlon 64 X2 Dual Core with 2.5 GHz.

Model	LOC	|P|	|T|	|E|	t[s]
GSM	84	131	263	526	1.55
HTS_P	194	903	1103	3482	3.24
HTS_C	195	1912	3515	11881	15.7

References

1. Abdulla, P.A., Bouajjani, A., Cederberg, J., Haziza, F., Rezine, A.: Monotonic abstraction for programs with dynamic memory heaps. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 341–354. Springer, Heidelberg (2008)
2. Abdulla, P.A., Čerans, K., Jonsson, B., Tsay, Y.-K.: Algorithmic analysis of programs with well quasi-ordered domains. Inf. Comp. 160(1-2), 109–127 (2000)
3. Busi, N., Gorrieri, R.: Distributed semantics for the π-calculus based on Petri nets with inhibitor arcs. JLAP 78(1), 138–162 (2009)
4. Devillers, R., Klaudel, H., Koutny, M.: A compositional Petri net translation of general π-calculus terms. For Asp. Comp. 20(4-5), 429–450 (2008)
5. Finkel, A., Schnoebelen, P.: Well-structured transition systems everywhere! TCS 256(1-2), 63–92 (2001)
6. HAL, http://fmt.isti.cnr.it:8080/hal/
7. Hirschkoff, D.: An extensional spatial logic for mobile processes. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 325–339. Springer, Heidelberg (2004)
8. Joshi, S., König, B.: Applying the graph minor theorem to the verification of graph transformation systems. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 214–226. Springer, Heidelberg (2008)
9. Khomenko, V., Koutny, M., Niaouris, A.: Applying Petri net unfoldings for verification of mobile systems. In: MOCA, Bericht FBI-HH-B-267/06, pp. 161–178. University of Hamburg (2006)
10. Khomenko, V., Meyer, R.: Checking π-calculus structural congruence is graph isomorphism complete. In: ACSD, pp. 70–79. IEEE, Los Alamitos (2009)
11. König, B., Kozioura, V.: Counterexample-guided abstraction refinement for the analysis of graph transformation systems. In: Hermanns, H., Palsberg, J. (eds.) TACAS 2006. LNCS, vol. 3920, pp. 197–211. Springer, Heidelberg (2006)
12. König, B., Kozioura, V.: Incremental construction of coverability graphs. IPL 103(5), 203–209 (2007)
13. Model Checking Kit, http://www.fmi.uni-stuttgart.de/szs/tools/mckit/
14. Meyer, R.: A theory of structural stationarity in the π-calculus. Acta Inf. 46(2), 87–137 (2009)
15. Meyer, R., Gorrieri, R.: On the relationship between π-calculus and finite place/transition Petri nets. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 463–480. Springer, Heidelberg (2009)
16. Meyer, R., Khomenko, V., Strazny, T.: A practical approach to verification of mobile systems using net unfoldings. Fund. Inf. 94(3-4), 439–471 (2009)

17. NAUTY, http://cs.anu.edu.au/~bdm/nauty/
18. Orava, F., Parrow, J.: An algebraic verification of a mobile network. For. Asp. Comp. 4(6), 497–543 (1992)
19. Petruchio, http://petruchio.informatik.uni-oldenburg.de
20. Rensink, A.: Canonical graph shapes. In: Schmidt, D. (ed.) ESOP 2004. LNCS, vol. 2986, pp. 401–415. Springer, Heidelberg (2004)
21. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. ACM TOPLAS 24(3), 217–298 (2002)
22. 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)
23. SPATIAL LOGIC MODEL CHECKER, http://ctp.di.fct.unl.pt/SLMC/
24. Wies, T., Zuffrey, D., Henzinger, T.A.: Forward analysis of depth-bounded processes. In: Ong, L. (ed.) FOSSACS 2010. LNCS, vol. 6014, pp. 94–108. Springer, Heidelberg (2010)

Synthesis of Quantized Feedback Control Software for Discrete Time Linear Hybrid Systems

Federico Mari, Igor Melatti, Ivano Salvo, and Enrico Tronci

Dip. di Informatica Sapienza Università di Roma,
Via Salaria 113, 00198 Roma, Italy
{mari,melatti,salvo,tronci}@di.uniroma1.it

Abstract. We present an algorithm that given a *Discrete Time Linear Hybrid System* $\mathcal{H}$ returns a correct-by-construction software implementation K for a (*near time optimal*) robust quantized feedback controller for $\mathcal{H}$ along with the set of states on which K is guaranteed to work correctly (*controllable region*). Furthermore, K has a *Worst Case Execution Time* linear in the number of bits of the quantization schema.

1 Introduction

Software generation from models and formal specifications forms the core of model based design of embedded software [18]. This approach is particularly interesting for control systems since in such a case system specifications are much easier to define than the control software behavior itself.

A *control system* consists of two subsystems (forming the *closed loop* system): the *controller* and the *plant*. In an endless loop the controller measures outputs from and sends commands to the plant in order to drive it towards a given *goal*. In our setting the controller consists of software implementing the *control law*. System requirements are typically given as specifications for the closed loop system. Control engineering techniques are used to design the *control law* (i.e. the functional specifications for the control software) from the closed loop system specifications. Software engineering techniques are then used to design *control software* implementing a given control law.

Unfortunately, when the plant model is a *hybrid system* [4,1,3] existence of a control law is undecidable (e.g. see [17]) even for linear hybrid automata. This scenario is further complicated by the *quantization* process always present in software based control systems. Namely, measures from *sensors* go through an AD (analog-to-digital) conversion before being sent to the control software and commands from the control software go through a DA (digital-to-analog) conversion before being sent to plant *actuators*. Furthermore, typically a *robust* control is desired, that is, one that meets the given closed loop requirements notwithstanding (nondeterministic) variations in the plant parameters.

As for hybrid systems, no approach is available for the automatic synthesis of robust quantized feedback control laws *and* of their software implementation. This motivates the focus of our paper.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 180–195, 2010.

Our Main Contributions. A *Discrete Time Linear Hybrid System* (DTLHS) is a discrete time hybrid system whose dynamics is defined as the logical conjunction of linear constraints on its continuous as well as discrete variables.

We present an effective algorithm that, given a DTLHS model $\mathcal{H}$ for the plant and a quantization schema (i.e. how many bits we use for AD conversion), returns a pair (K, R), where: K is a correct-by-construction software implementation (C language in our case) of a (*near time optimal*) *Quantized Feedback Controller* (QFC) for $\mathcal{H}$ and R is an OBDD [10] representation of the set of states (*controllable region*) on which K is guaranteed to meet the closed loop requirements. Furthermore, K is *robust* with respect to nondeterministic variations in the plant parameters and has a *Worst Case Execution Time* (WCET) guaranteed to be linear in the number of bits of the quantization schema.

We implemented our algorithm on top of the CUDD package and of the GLPK *Mixed Integer Linear Programming* (MILP) solver and present experimental results on using our tool to synthesize robust QFCs for a widely used mixed-mode analog circuit: the buck DC-DC converter (e.g. see [26]).

Analog DC-DC converters are a vital part of many mission (e.g. satellites) or safety (e.g. aircrafts) critical applications. However the ever increasing demand for energy efficiency and easy reconfigurability makes fully software based switching converters (e.g., as in [26]) a very attractive alternative to analog ones. Unfortunately, lack of formal reliability assessment (in order to bound the failure probability to 10^{-9}) limits the deployment of software based switching converters in safety critical applications. Reliability analysis for switching converters using an analog control schema has been studied in [13]. For software based converters, carrying out such a reliability analysis entails formal verification of the control law as well as of its software implementation. The above considerations make the buck DC-DC converter a very interesting (and challenging) example for automatic synthesis of correct-by-construction control software.

Our experimental results show that within about 20 hours of CPU time and within 200MB of RAM we can synthesize (K, R) as above for a 10 bit quantized buck DC-DC converter.

Related Work. Synthesis of *Quantized Feedback Control Laws* for linear systems has been widely studied in control engineering (e.g. see [14]). However, to the best of our knowledge, no previously published paper addresses synthesis of *Quantized Feedback Control Software* for DTLHSs. Indeed, our work differs from previously published ones in the following aspects: (1) we provide a *tool* for automatic synthesis of correct-by-construction control software (rather than *design methodologies* for the control law); (2) we synthesize *robust control software* (thus encompassing quantization) whereas robust control law design techniques do not take into account the software implementation; (3) in order to generate provably correct software, we assume a nondeterministic (*malicious*) model for quantization errors rather than a stochastic one, as usually done in control engineering; (4) our synthesis tool also returns the *controllable region*, that is the set of states on which the synthesized control software is guaranteed to work correctly (this is very important for *Fault Detection Isolation and Recovery*, FDIR, e.g. see [21]). In the following we discuss some related literature.

Quantization can be seen as a form of abstraction, where the abstract state space and transitions are defined by the number of bits of AD conversion. Abstraction for hybrid systems has been widely studied. For example, see [25,2,20,19] and citations thereof. Note however that all published literature on abstraction focuses on designing abstractions to support verification or control law design. In our case instead, the abstraction is fully defined by the AD conversion schema and our focus is on devising techniques to effectively remove abstract transitions in order to counteract the nondeterminism (information loss) stemming from the quantization process.

Control synthesis for *Timed Automata* (TA) [4], *Linear Hybrid Automata* (LHA) [1,3] as well as nonlinear hybrid systems has been extensively studied. Examples are in [22,11,6,30,16,28,5,9,8] and citations thereof. We note however that all above papers address design of control laws and do not take into account the quantization process, that is, they assume *exact* (i.e. real valued) state measures. Here instead we address design of quantized feedback control software.

Correct-by-construction software synthesis in a finite state context has been studied in [7,29,27,12]. The above approaches cannot be directly used in our context since they do not account for continuous state variables.

2 Background

Unless otherwise stated each variable x ranges on a known bounded interval $\mathcal{D}_x$ either of the reals or of the integers (discrete variables). We denote with $\sup(x)$ ($\inf(x)$) the sup (inf) of $\mathcal{D}_x$. Boolean variables are discrete variables ranging on the set $\mathbb{B} = \{0, 1\}$. We denote with $X = [x_1, \ldots x_n]$ a finite sequence (list) of variables, with $\cup$ list concatenation and with $\mathcal{D}_X = \prod_{x \in X} \mathcal{D}_x$ the domain of X.

A *valuation* $X^* \in \mathcal{D}_X$ over a list of variables X is a function v that maps each variable $x \in X$ to a value $v(x)$ in $\mathcal{D}_x$. We may use the same notation to denote a variable (a syntactic object) and one of its valuations. The intended meaning will be always clear from the context. To clarify that a variable [valuation] x is real (integer, boolean) valued we may write x^r (x^d, x^b). Analogously X^r (X^d, X^b) denotes the sequence of real (integer, boolean) variables [valuations] in X. If x is a boolean variable [valuation] we write $\bar{x}$ for $(1 - x)$.

A *linear expression* (over X) is a linear combination with real coefficients of variables in X. A *constraint* (over X) is an expression of the form $\alpha \bowtie b$ where α is a linear expression over X, $\bowtie$ is one of $\leq, \geq, =$ and b is a real constant. A constraint is a *predicate* on X. If $A(X)$ and $B(X)$ are *predicates* on X, then $(A(X) \wedge B(X))$ and $(A(X) \vee B(X))$ are *predicates* on X. A *conjunctive predicate* is just a conjunction of linear constraints. A *satisfying assignment* to $P(X)$ is a valuation X^* such that $P(X^*) = 1$. Abusing notation we may denote with P the set of satisfying assignments to $P(X)$. Given a predicate $P(X)$ and a fresh boolean variable $y \notin X$, the *if-then predicate* $y \rightarrow P(X)$ $[\bar{y} \rightarrow P(X)]$ denotes the predicate $((y = 0) \vee P(X))$ $[((y = 1) \vee P(X))]$. In our setting (bounded variables), for any predicate $P(X)$ there exists a sequence Z of fresh boolean variables and a conjunctive predicate $Q(Z, X)$ s.t. $\forall X[P(X) \iff \exists Z\, Q(Z, X)]$ (see [23] for details). Thus, any if-then predicate can be transformed into a conjunctive predicate. Accordingly, we will regard and use if-then predicates as conjunctive predicates.

A *Mixed Integer Linear Programming* (MILP) problem with *decision variables* X is a tuple $(\max,\ J(X),\ A(X))$ where: X is a list of variables, $J(X)$ (*objective function*) is a linear expression on X and $A(X)$ (*constraints*) is a conjunctive predicate on X. A solution to $(\max, J(X), A(X))$ is a valuation X^* s.t. $A(X^*)$ holds and, for any valuation Ξ, $(A(\Xi) \rightarrow (J(\Xi) \leq J(X^*)))$. We write $(\min, J(X), A(X))$ for $(\max, -J(X), A(X))$. A *feasibility* problem is a MILP problem of the form $(\max, 0, A(X))$. We write also $A(X)$ for $(\max, 0, A(X))$.

A *Labeled Transition System* (LTS) is a tuple $\mathcal{S} = (S, A, T)$ where: S is a (possibly infinite) set of states, A is a (possibly infinite) set of *actions*, $T : S \times A \times S \rightarrow \mathbb{B}$ is the *transition relation* of $\mathcal{S}$. Let $s \in S$ and $a \in A$. We denote with: $\mathrm{Adm}(\mathcal{S}, s)$ the set of actions admissible in s, that is $\mathrm{Adm}(\mathcal{S}, s) = \{a \in A \mid \exists s' T(s, a, s')\}$ and with $\mathrm{Img}(\mathcal{S}, s, a)$ the set of next states from s via a, that is $\mathrm{Img}(\mathcal{S}, s, a) = \{s' \in S \mid T(s, a, s')\}$. A *run* or *path* for $\mathcal{S}$ is a sequence $\pi = s(0)a(0)s(1)a(1)s(2)a(2)\ldots$ of states $s(t)$ and actions $a(t)$ such that $\forall t \geq 0$ $T(s(t), a(t), s(t+1))$. The length $|\pi|$ of a run π is the number of actions in π. We denote with $\pi^{(S)}(t)$ the t-th state element of π, and with $\pi^{(A)}(t)$ the t-th action element of π. That is $\pi^{(S)}(t) = s(t)$, and $\pi^{(A)}(t) = a(t)$.

3 Discrete Time Linear Hybrid Systems

In this section we introduce *Discrete Time Linear Hybrid Systems* (DTLHS).

Definition 1. *A* Discrete Time Linear Hybrid System *(DTLHS) is a tuple* $\mathcal{H} = (X, U, Y, N)$ *where:*

- $X = X^r \cup X^d$ *is a finite sequence of real* (X^r) *and discrete* (X^d) present state *variables. We denote with* X' *the sequence of* next state *variables obtained by decorating with* $'$ *all variables in* X.
- $U = U^r \cup U^d$ *is a finite sequence of* input *variables.*
- $Y = Y^r \cup Y^d$ *is a finite sequence of* auxiliary *variables. Auxiliary variables are typically used to model* modes *(e.g., from switching elements such as diodes) or* uncontrollable inputs *(e.g., disturbances).*
- $N(X, U, Y, X')$ *is a* conjunctive predicate *over* $X \cup U \cup Y \cup X'$ *defining the transition relation (*next state*) of the system.*

Note that in our setting (bounded variables) any predicate can be transformed into a conjunctive predicate (Sect. 2). Accordingly, in Def. 1, without loss of generality we focused on conjunctive predicates in order to simplify our exposition.

The dynamics of a DTLHS $\mathcal{H} = (X, U, Y, N)$ is defined by $\mathrm{LTS}(\mathcal{H}) = (\mathcal{D}_X, \mathcal{D}_U, \bar{N})$ where: $\bar{N} : \mathcal{D}_X \times \mathcal{D}_U \times \mathcal{D}_X \rightarrow \mathbb{B}$ is a function s.t. $\bar{N}(s, a, s') = \exists\, y \in \mathcal{D}_Y\ N(s, a, y, s')$. A *state* for $\mathcal{H}$ is a state for $\mathrm{LTS}(\mathcal{H})$ and a *run* (or *path*) for $\mathcal{H}$ is a run for $\mathrm{LTS}(\mathcal{H})$ (Sect. 2).

Example 1. Let $\mathcal{H} = (\{x\}, \{u\}, \varnothing, N)$ with $\mathcal{D}_x = [-2.5, 2.5]$, $\mathcal{D}_u = \{0, 1\}$, and $N(x, u, x') = [\overline{u} \rightarrow x' = \alpha x] \wedge [u \rightarrow x' = \beta x]$ with $\alpha = \frac{1}{2}$ and $\beta = \frac{3}{2}$. When $Y = \varnothing$ (as here) for the sake of simplicity we omit it from N arguments.

Adding nondeterminism to $\mathcal{H}$ allows us to synthesize *robust* controllers. For example, variations in the parameter α can be modelled with a tolerance $\rho \in [0, 1]$

(e.g., $\rho = 0.5$) for α. This replaces N with: $N^\rho = [\overline{u} \rightarrow x' \leq (1+\rho)\alpha x] \wedge [\overline{u} \rightarrow x' \geq (1-\rho)\alpha x] \wedge [u \rightarrow x' = \beta x]$. Suitable control synthesis on $\mathcal{H}^\rho = (\{x\}, \{u\}, \varnothing, N^\rho)$ will yield a *robust* (up to ρ) controller for $\mathcal{H}$.

Example 2. The buck DC-DC converter (right part of Fig. 1) is a mixed-mode analog circuit converting the DC input voltage (V_i in Fig. 1) to a desired DC output voltage (v_O in Fig. 1). The typical software based approach (e.g. see [26]) is to control the switch u in Fig. 1 (typically implemented with a MOSFET) with a microcontroller. Designing the software to run on the microcontroller to properly actuate the switch is the control design problem for the buck DC-DC converter in our context. The circuit in Fig. 1 can be modeled as a DTLHS $\mathcal{H} = (X, U, Y, N)$ with: $X = X^r = [i_L, v_O]$, $U = U^d = [u]$, $Y = Y^r \cup Y^d$ with $Y^r = [i_u, v_u, i_D, v_D]$ and $Y^d = [q]$. $\mathcal{H}$ auxiliary variables Y stem from the constitutive equations of the switching elements (i.e. the switch u and the diode D in Fig. 1). The transition relation $N(X, U, Y, X')$ for $\mathcal{H}$ is shown in Fig. 1 (left) where we use a discrete time model with sampling time T (writing x' for $x(t+1)$) and model a tolerance $\rho = 0.25$ (25%) on V_i values. In Fig. 1 (left), constants $a_{i,j}, b_{i,j}$ depend on the circuit parameters R, r_L, r_C, L, C and algebraic constraints (i.e. constraints not involving next state variables) stem from the constitutive equations of the switching elements (see [23] for details).

$$
\begin{aligned}
&N(X,U,Y,X') = ((i_L{}' = (1 + Ta_{1,1})i_L + Ta_{1,2}v_O + \\
&Tb_{1,1}v_D) \\
&\wedge\ (v_O{}' = Ta_{2,1}i_L + (1 + Ta_{2,2})v_O + Tb_{2,1}v_D) \\
&\wedge\ (v_u - v_D \leq (1+\rho)V_i) \wedge (v_u - v_D \geq (1-\rho)V_i) \\
&\wedge\ (i_D = i_L - i_u) \wedge (q \rightarrow v_D = 0) \wedge (q \rightarrow i_D \geq 0) \\
&\wedge\ (\bar{q} \rightarrow v_D \leq 0) \wedge (\bar{q} \rightarrow v_D = R_{off} i_D) \\
&\wedge\ (u \rightarrow v_u = 0) \wedge (\bar{u} \rightarrow v_u = R_{off} i_u))
\end{aligned}
$$

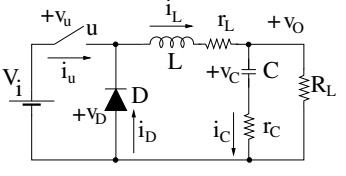


Fig. 1. Buck DC-DC converter

4 Quantized Feedback Control Problem for DTLHS

We define the *Feedback Control Problem* for LTSs (Def. 2) and for DTLHSs (Def. 4). On such a base we define the *Quantized Feedback Control Problem* for DTLHSs (Def. 6).

We begin by extending to (possibly infinite) LTSs the definitions in [29,12] for finite LTSs. In what follows, let $\mathcal{S} = (S, A, T)$ be an LTS, I, $G \subseteq S$ be, respectively, the *initial* and *goal* sets of $\mathcal{S}$.

Definition 2. *A* controller *for $\mathcal{S}$ is a function $K : S \times A \rightarrow \mathbb{B}$ s.t. $\forall s \in S$, $\forall a \in A$, if $K(s,a)$ then $\exists s'\ T(s,a,s')$. $\mathrm{Dom}(K)$ denotes the set of states for which at least a control action is enabled. Formally,* $\mathrm{Dom}(K) = \{s \in S \mid \exists a\ K(s,a)\}$. *$\mathcal{S}^{(K)}$ denotes the* closed loop system*, that is the LTS $(S, A, T^{(K)})$, where $T^{(K)}(s,a,s') = T(s,a,s') \wedge K(s,a)$. A* control problem *for $\mathcal{S}$ is a triple $(\mathcal{S}, I, G)$.*

A *controller* for $\mathcal{S}$ (Def. 2) is used to restrict $\mathcal{S}$ behavior so that all states in the initial region (I) will reach in one or more steps the goal region (G). In the following, we formalize such a concept by defining strong and weak solutions to an LTS control problem.

We call a path π *fullpath* [7] if either it is infinite or its last state $\pi^{(S)}(|\pi|)$ has no successors. We denote with $\mathrm{Path}(s)$ the set of fullpaths starting in state s, i.e. the set of fullpaths π such that $\pi^{(S)}(0) = s$. Observe that $\mathrm{Path}(s)$ is never empty, since it contains at least the path of length 0 containing only state s.

Given a path π in $\mathcal{S}$, $J(\mathcal{S}, \pi, G)$ denotes the unique $n > 0$, if it exists, s.t. $[\pi^{(S)}(n) \in G] \wedge [\forall\, 0 < i < n . \pi^{(S)}(i) \notin G]$, $+\infty$ otherwise. We require $n > 0$ since our systems are nonterminating and each controllable state (including a goal state) must have a path of positive length to a goal state. The *worst case distance* (*pessimistic view*) of a state s from the goal region G is $J_{strong}(\mathcal{S}, G, s) = \sup\{J(\mathcal{S}, \pi, G) \mid \pi \in \mathrm{Path}(s)\}$. The *best case distance* (*optimistic view*) of a state s from the goal region G is $J_{weak}(\mathcal{S}, G, s) = \inf\{J(\mathcal{S}, \pi, G) \mid \pi \in \mathrm{Path}(s)\}$.

Definition 3. *A* strong [weak] *solution to a control problem* $\mathcal{P} = (\mathcal{S},\ I,\ G)$ *is a controller* K *for* $\mathcal{S}$*, such that* $I \subseteq \mathrm{Dom}(K)$ *and for all* $s \in \mathrm{Dom}(K)$*,* $J_{strong}(\mathcal{S}^{(K)}, G, s)$ $[J_{weak}(\mathcal{S}^{(K)}, G, s)]$ *is finite.*

Example 3. Let $\mathcal{S}_0$ $[\mathcal{S}_1]$ be the LTS which transition relation consists of the continuous [all] arrows in Fig. 2 (left). Let $\hat{I} = \{-1, 0, 1\}$ and $\hat{G} = \{0\}$. Then, $\hat{K}(\hat{s}, \hat{u}) \equiv [\hat{s} \neq 0 \Rightarrow \hat{u} = 0]$ is a strong solution to the control problem $(\mathcal{S}_0,\ \hat{I},\ \hat{G})$ and a weak solution to $(\mathcal{S}_1,\ \hat{I},\ \hat{G})$.

Remark 1. Note that if K is a strong solution to $(\mathcal{S},\ I,\ G)$ and $G \subseteq I$ (as it is usually the case in control problems) then all paths starting from $\mathrm{Dom}(K)$ $(\subseteq I)$ will *touch* G infinitely often (*stability*).

A DTLHS control problem is a triple $(\mathcal{H}, I, G)$ where $\mathcal{H}$ is a DTLHS and $(\mathrm{LTS}(\mathcal{H}),\ I,\ G)$ is an LTS control problem. For DTLHSs we restrict ourselves to control problems where I and G can be represented as conjunctive predicates. From [17] it is easy to show that DTLHS control problems are undecidable [23]. For DTLHS control problems usually *robust* controllers are desired. That is, controllers that, notwithstanding nondeterminism in the plant (e.g. due to parameter variations), drive the plant state to the goal region. For this reason, and to counteract the nondeterminism stemming from the quantization process, we focus on strong solutions. Furthermore, to accommodate quantization errors, always present in software based controllers, it is useful to relax the notion of control solution by tolerating an (arbitrarily small) error ε on the continuous variables. This leads to the definition of ε-solution. Let ε be a nonnegative real number, $W^r = \prod_{i=1}^{n} W_i^r \subseteq \mathcal{D}_X^r$, $W^d = \prod_{i=1}^{m} W_i^d \subseteq \mathcal{D}_X^d$ and $W = W^r \times W^d \subseteq \mathcal{D}_X^r \times \mathcal{D}_X^d$. The *$\varepsilon$-relaxation* of W is the set (*ball* of radius ε) $\mathcal{B}_\varepsilon(W) = \{(z_1, \ldots z_n, q_1, \ldots q_m) \mid (q_1, \ldots q_m) \in W^d$ and $\forall i \in \{1, \ldots n\}\ \exists\, x_i \in W_i^r$ s.t. $|z_i - x_i| \leq \varepsilon\}$.

Definition 4. *Let* $(\mathcal{H}, I, G)$ *be a DTLHS control problem and* ε *be a nonnegative real number. An* ε*-solution to* $(\mathcal{H}, I, G)$ *is a* strong solution *to the LTS control problem* $(\mathrm{LTS}(\mathcal{H}), I, \mathcal{B}_\varepsilon(G))$.

Example 4. Let $\mathcal{P} = (\mathcal{H}, I, G)$, $\mathcal{H}$ as in Ex. 1, $I = \mathcal{D}_x$ and $G = \{0\}$ (represented by conjunctive predicate $x = 0$). Control problem $\mathcal{P}$ has no solution (because of the Zeno phenomenon), but for all $\varepsilon > 0$ it has the ε-solution K s.t. $\forall x \in I.\ K(x, 0) = 1$.

Example 5. The typical goal of a controller for the buck DC-DC converter in Ex. 2 is keeping the output voltage v_O *close enough* to a given reference value V_{ref}. This leads to the control problem $\mathcal{P} = (\mathcal{H}, I, G)$ where: $\mathcal{H}$ is defined in Ex. 2, $I = (|i_L| \leq 2) \wedge (0 \leq v_O \leq 6.5)$, $G = (|v_O - V_{ref}| \leq \theta) \wedge (|i_L| \leq 2)$ and $\theta = 0.01$ is the desired converter precision.

In order to define quantized feedback control problems for DTLHSs (Def. 6) we introduce *quantizations* (Def. 5). Let x be a real valued variable ranging on a bounded interval of reals $\mathcal{D}_x = [a_x, b_x]$. A *quantization* for x is a function γ from $\mathcal{D}_x$ to a bounded interval of integers $\gamma(\mathcal{D}_x) = [\hat{a_x}, \hat{b_x}]$. For ease of notation we extend quantizations to integer variables ranging on a bounded interval of integers by stipulating that the only quantization γ for such variables is the identity function (i.e. $\gamma(x) = x$). The *width* $\left\|\gamma^{-1}(v)\right\|$ of $v \in \gamma(\mathcal{D}_x)$ in γ is defined as follows: $\left\|\gamma^{-1}(v)\right\| = \sup\{\ |w - z| \mid w, z \in \mathcal{D}_x \ \wedge\ \gamma(w) = \gamma(z) = v\}$. The *quantization step* $\|\gamma\|$ is defined as follows: $\|\gamma\| = \max\left\{\ \left\|\gamma^{-1}(v)\right\| \mid v \in \gamma(\mathcal{D}_x)\right\}$.

Definition 5. *Let $\mathcal{H} = (X, U, Y, N)$ be a DTLHS. A* quantization *Γ for $\mathcal{H}$ is a set of maps $\Gamma = \{\gamma_w \mid \gamma_w$ is a quantization for $w \in X \cup U\}$. Let $W = [w_1, \ldots w_k] \subseteq X \cup U$ and $v = [v_1, \ldots v_k] \in \mathcal{D}_W$. We write $\Gamma(v)$ for the tuple $[\gamma_{w_1}(v_1), \ldots \gamma_{w_k}(v_k)]$ and $\Gamma(\mathcal{D}_W)$ for the set of tuples $\{\Gamma(v) \mid v \in \mathcal{D}_W\}$. Finally, the* quantization step *$\|\Gamma\|$ for Γ is defined as: $\|\Gamma\| = \max\{\ \|\gamma\| \mid \gamma \in \Gamma\}$.*

A control problem admits a *quantized* solution if control decisions can be made by just looking at quantized values. This enables a software implementation for a controller.

Definition 6. *Let $\mathcal{H} = (X, U, Y, N)$ be a DTLHS, Γ be a quantization for $\mathcal{H}$ and $\mathcal{P} = (\mathcal{H}, I, G)$ be a control problem. A Γ* Quantized Feedback Control *(QFC) solution to $\mathcal{P}$ is a $\|\Gamma\|$-solution $K(x, u)$ to $\mathcal{P}$ such that $K(x, u) = \hat{K}(\Gamma(x), \Gamma(u))$ where $\hat{K} : \Gamma(\mathcal{D}_X) \times \Gamma(\mathcal{D}_U) \to \mathbb{B}$.*

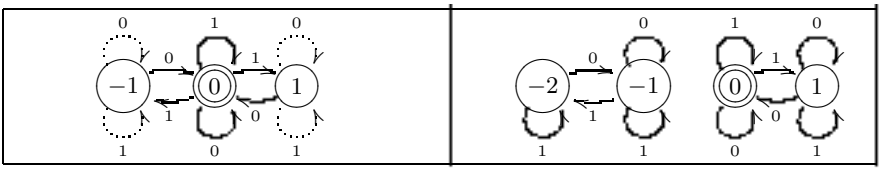


Fig. 2. Γ control abstraction for DTLHSs in Exs. 3 and 8 (left) and Ex. 9 (right)

Example 6. Let $\mathcal{P}$ be as in Ex. 4, $\Gamma(x) = round(x/2)$ (where $round(x) = \lfloor x \rfloor + \lfloor 2(x - \lfloor x \rfloor) \rfloor$ is the usual rounding function) and $\hat{K}$ as in Ex. 3. Then, $\|\Gamma\| = 2$ and $K(x, u) = \hat{K}(\Gamma(x), \Gamma(u))$ is a Γ QFC solution to $\mathcal{P}$.

5 Control Abstraction

The AD process maps intervals of state values into discrete state values. As a result the control software *sees* the controlled system (plant) as a nondeterministic finite automaton. Of course we want our control software to work notwithstanding such a nondeterminism (strong solution, Def. 3). To this end we should try to *limit* such a nondeterminism as much as possible. This leads to the notion of *control abstraction* (Def. 8), the main focus of this section.

Since QFC (Def. 6) rests on AD conversion we must be careful not to drive the plant outside the bounds in which AD conversion works correctly. This leads to the definition of *safe action* (Def. 7). Intuitively, an action is safe in a state if it never drives the system outside of its state bounds.

Definition 7. *Let $\mathcal{H} = (X, U, Y, N)$ be a DTLHS and Γ be a quantization.*

1. *We say that action $u \in \mathcal{D}_U$ is* safe *for $s \in \mathcal{D}_X$ (in $\mathcal{H}$) if for all s', $[\exists y \in \mathcal{D}_Y\ N(s, u, y, s')$ implies $s' \in \mathcal{D}_X]$.*
2. *We say that action $\hat{u} \in \Gamma(\mathcal{D}_U)$ is Γ-safe in state $\hat{s} \in \Gamma(\mathcal{D}_X)$ if for all $s \in \Gamma^{-1}(\hat{s})$, $u \in \Gamma^{-1}(\hat{u})$, u is safe for s in $\mathcal{H}$.*

Note that, in general, not all actions $u \in \mathcal{D}_U$ are safe in $\mathcal{H}$ since Def. 1 only asks N to be a conjunctive predicate.

Example 7. Let $\mathcal{H}$ be as in Ex. 1. Then action $u = 1$ is not safe in state $s = 2$ since we have $N(2, 1, 3)$, and $s' = 3$ is outside $\mathcal{H}$ state bounds.

A *control abstraction* (Def. 8) is a finite state automaton modelling how a DTLHS is *seen* from the control software because of AD conversion.

Definition 8. *Let $\mathcal{H} = (X, U, Y, N)$ be a DTLHS and Γ be a quantization for $\mathcal{H}$. We say that the LTS $\hat{\mathcal{H}} = (\Gamma(\mathcal{D}_X), \Gamma(\mathcal{D}_U), \hat{N})$ is a Γ* control abstraction *of $\mathcal{H}$ if its transition relation $\hat{N}$ satisfies the following conditions.*

1. *Each abstract transition stems from a concrete transition. Formally: for all $\hat{s}, \hat{s}' \in \Gamma(\mathcal{D}_X)$, $\hat{u} \in \Gamma(\mathcal{D}_U)$, if $\hat{N}(\hat{s}, \hat{u}, \hat{s}')$ then there exist $s \in \Gamma^{-1}(\hat{s})$, $u \in \Gamma^{-1}(\hat{u})$, $s' \in \Gamma^{-1}(\hat{s}')$, $y \in \mathcal{D}_Y$ s.t. $N(s, u, y, s')$.*
2. *If an abstract action is safe then all its possible concrete effects (besides self-loops) are faithfully represented in the abstract system. Formally: for all $\hat{s} \in \Gamma(\mathcal{D}_X)$, $\hat{u}$ Γ-safe in $\hat{s}$, $s \in \Gamma^{-1}(\hat{s})$, $u \in \Gamma^{-1}(\hat{u})$, $s' \in \mathcal{D}_X$, if $[\exists y \in \mathcal{D}_Y\ N(s, u, y, s')]$ and $\Gamma(s) \neq \Gamma(s')$ then $\hat{N}(\Gamma(s), \Gamma(u), \Gamma(s'))$.*
3. *If there is no upper bound to the length of concrete paths inside the counter-image of an abstract state then there is an (abstract) self-loop. Formally: for all $\hat{s} \in \Gamma(\mathcal{D}_X)$, $\hat{u} \in \Gamma(\mathcal{D}_U)$, if $\forall k\ \exists x(0), \ldots x(k+1) \in \Gamma^{-1}(\hat{s})$ $\exists u(0), \ldots u(k) \in \Gamma^{-1}(\hat{u})\ \exists y(0), \ldots y(k) \in \mathcal{D}_Y\ [\bigwedge_{t=0}^{k} N(x(t), u(t), y(t), x(t+1))]$ then $\hat{N}(\hat{s}, \hat{u}, \hat{s})$.*

We say that $\hat{\mathcal{H}}$ is a control abstraction *of $\mathcal{H}$ if $\hat{\mathcal{H}}$ is a Γ* control abstraction *of $\mathcal{H}$ for some quantization Γ. Finally, we denote with $\mathcal{A}_\Gamma(\mathcal{H})$ the set of all Γ control abstractions on $\mathcal{H}$.*

Note that any abstraction (e.g. see [2]) is also a control abstraction. However, the converse is false since some concrete transition (e.g. a self loop or an unsafe action) may have no abstract image. Let $\mathcal{S}_1 = (S, A, T_1)$ and $\mathcal{S}_2 = (S, A, T_2)$ be LTSs. We say that $\mathcal{S}_1$ *refines* $\mathcal{S}_2$ (notation $\mathcal{S}_1 \sqsubseteq \mathcal{S}_2$) iff for each $s, s' \in S$, $a \in A$, $T_1(s, a, s')$ implies $T_2(s, a, s')$. The binary relation $\sqsubseteq$ is a partial order. Moreover, the poset $(\mathcal{A}_\Gamma(\mathcal{H}), \sqsubseteq)$ is a *lattice*. Furthermore, since $\mathcal{A}_\Gamma(\mathcal{H})$ is a finite set, the poset $(\mathcal{A}_\Gamma(\mathcal{H}), \sqsubseteq)$ has unique *maximum* and unique *minimum* elements.

Example 8. Let $\mathcal{H}$ be as in Ex. 1 and Γ be as in Ex. 6. Each Γ control abstraction of $\mathcal{H}$ has the form $\hat{\mathcal{H}} = (\{-1, 0, 1\}, \{0, 1\}, \hat{N})$, where the set of transitions in $\hat{N}$ is any subset, containing all continuous arrows, of the set of transitions of the automaton depicted in Fig. 2 (left). In particular, a control abstraction may omit some self loops (namely, those with dotted arrows in Fig. 2). Transitions $\hat{N}(0, 0, 0)$ and $\hat{N}(0, 1, 0)$ must belong to all Γ control abstractions, because of condition 3 in Def. 8. In fact all paths starting in 0 will remain in 0 forever. The transition relation defined in Fig. 2 (left) by continuous arrows is the minimum Γ control abstraction $\hat{\mathcal{H}}_{min}$ of $\mathcal{H}$ whereas the transition relation defined by all arrows is the maximum Γ control abstraction $\hat{\mathcal{H}}_{max}$ of $\mathcal{H}$. Note that there is no controller (strongly) driving all states of $\hat{\mathcal{H}}_{max}$ to state 0. In fact, because of self-loops, action 0 from state 1 may lead to state 0 as well as to state 1 (self-loop). On the other hand the controller $\hat{K}$ enabling only action 0 in any state will (weakly) drive all states of $\hat{\mathcal{H}}_{max}$ to 0 since each state in $\hat{\mathcal{H}}_{max}$ has at least a 0-labelled transition leading to state 0. Controller $\hat{K}$ will also (strongly and thus weakly) drive all states of $\hat{\mathcal{H}}_{min}$ (including 0) to state 0.

Remark 2. Example 8 suggests that we should focus on minimum control abstractions in order to increase our chances of finding a strong controller. Correctness of such an intuition will be shown in Theor. 1. As for computing the minimum control abstraction we note that this entails deciding if a given self-loop can be eliminated according to condition 3 in Def. 8. Unfortunately it is easy to show that such a problem comes down to solve a reachability problem on linear hybrid systems, that, by [17], is undecidable. Thus, self-loop eliminability is undecidable too in our context. As a result, in general, we cannot hope to compute *exactly* the minimum control abstraction.

6 Synthesis of Quantized Feedback Control Software

We outline our synthesis algorithm QKS (*Quantized feedback Kontrol Synthesis*) and give its properties (Theor. 1). Details are in [23]. QKS takes as input a tuple $(\Gamma, \mathcal{H}, I, G)$, where: $\mathcal{H} = (X, U, Y, N)$ is a DTLHS, Γ is a quantization for $\mathcal{H}$ and $(\mathcal{H}, I, G)$ is a control problem. QKS returns a tuple $(\mu, \hat{D}, \hat{K})$, where: $\mu \in \{\textsc{Sol}, \textsc{NoSol}, \textsc{Unk}\}$, $K(x, u) = \hat{K}(\Gamma(x), \Gamma(u))$ is a Γ QFC solution for $\mathcal{H}$ (Def. 6), $\hat{D} = \mathrm{Dom}(\hat{K})$ and $D = \Gamma^{-1}(\hat{D}) = \mathrm{Dom}(K)$ is K controllable region.

We compute QKS output as follows. As a first step we compute a Γ control abstraction $\hat{\mathcal{Q}} = (\Gamma(\mathcal{D}_X), \Gamma(\mathcal{D}_U), \hat{N})$ of $\mathcal{H}$ as close as we can (see Remark 2) to the minimum one. Sect. 6.1 (function minCtrAbs in Alg. 1) outlines how $\hat{\mathcal{Q}}$ can be

computed. Let $\hat{I} = \Gamma(I)$, $\hat{G} = \Gamma(G)$ and $\hat{K}$ be the *most general optimal* (mgo) strong solution to the (LTS) control problem $(\hat{\mathcal{Q}}, \varnothing, \hat{G})$. Intuitively, the mgo strong solution $\hat{K}$ to a control problem $(\hat{\mathcal{Q}}, \varnothing, \hat{G})$ is the *unique* strong solution that, disallowing as few actions as possible, drives as many states as possible to a state in $\hat{G}$ along a shortest path. We compute (the OBDD representation for) $\hat{K}$ by implementing a suitable variant of the algorithm in [12]. Finally, we define: $K(x,u) = \hat{K}(\Gamma(x), \Gamma(u))$, $\hat{D} = \mathrm{Dom}(\hat{K})$, and $D = \Gamma^{-1}(\hat{D}) = \mathrm{Dom}(K)$.

If $\hat{I} \subseteq \hat{D}$ then QKS returns $\mu = \textsc{Sol}$. Note that in such a case, from the construction in [12], $\hat{K}$ is time optimal for the control problem $(\hat{\mathcal{Q}}, \hat{I}, \hat{G})$, thus K will *typically* move along a shortest path to G (i.e., K is *near time-optimal*). If $\hat{I} \not\subseteq \hat{D}$ then we compute the maximum Γ control abstraction $\hat{\mathcal{W}}$ of $\mathcal{H}$ and use the algorithm in [29] to check if there exists a weak solution to $(\hat{\mathcal{W}}, \hat{I}, \hat{G})$. If that is the case QKS returns $\mu = \textsc{Unk}$, otherwise QKS returns $\mu = \textsc{NoSol}$. Note that the maximum control abstraction may contain also (possibly) unsafe transitions (condition 2 of Def. 8). Thus a weak solution for $\hat{\mathcal{W}}$ may exist even when no weak solution for $\hat{\mathcal{Q}}$ exists. Using the above notations Theor. 1 summarizes the main properties of QKS.

Theorem 1. *Let $\mathcal{H}$ be a DTLHS, Γ be a quantization and $(\mathcal{H}, I, G)$ be a control problem. Then* QKS*$(\Gamma, \mathcal{H}, I, G)$ returns a triple $(\mu, \hat{D}, \hat{K})$ s.t.: $\mu \in \{\textsc{Sol}, \textsc{NoSol}, \textsc{Unk}\}$, $\hat{D} = \mathrm{Dom}(\hat{K})$, $D = \Gamma^{-1}(\hat{D})$ and $K = \hat{K}(\Gamma(x), \Gamma(u))$ is a Γ QFC solution to the control problem $(\mathcal{H}, D, G)$. Furthermore, the following holds.*

1. *If $\mu = \textsc{Sol}$ then $I \subseteq D$ and K is a Γ QFC solution to the control problem $(\mathcal{H}, I, G)$.*
2. *If $\mu = \textsc{NoSol}$ then there is no Γ QFC solution to the control problem $(\mathcal{H}, I, G)$.*
3. *If $\mu = \textsc{Unk}$ then* QKS *is inconclusive, that is $(\mathcal{H}, I, G)$ may or may not have a Γ QFC solution.*

Note that the AD conversion hardware is modelled by Γ and that from the OBDD for $\hat{K}$ above we get a C program (Section 6.2). Thus $\hat{K}$ as described above defines indeed the control software we are looking for. Finally, note that case 3 in Theor. 1 stems from undecidability of the QFC problem [17].

Example 9. Let $\mathcal{P} = (\mathcal{H}, I, G)$ be as in Ex. 4 and Γ be as in Ex. 8. For all Γ control abstractions $\hat{\mathcal{H}}$ (and thus for the minimum one shown in Ex. 8) not containing the self loops $\hat{N}(-1,0,-1)$ and $\hat{N}(1,0,1)$, $\hat{K}$ as in Ex. 3 is the mgo strong solution to $(\hat{\mathcal{H}}, \varnothing, \Gamma(G))$. Thus, $K(s,u)$ as in Ex. 6 is a Γ QFC solution to $\mathcal{P}$. Weak solutions to $(\hat{\mathcal{H}}, \Gamma(I), \Gamma(G))$ exist for all Γ control abstractions $\hat{\mathcal{H}}$. Note that existence of a Γ QFC solution to a control problem depends on Γ. Let us consider the quantization $\Gamma'(x)=\lfloor x/2 \rfloor$ for $\mathcal{H}$. Then the maximum Γ' control abstraction of $\mathcal{H}$ is $\mathcal{L} = (\{-2,-1,0,1\}, \{0,1\}, \hat{N})$, where the transition $\hat{N}$ is depicted in Fig. 2 (right). Clearly $(\mathcal{L}, \Gamma'(I), \Gamma'(G))$ has no weak solution since there is no path to the goal $\Gamma'(G) = \{0\}$ from any of the states -2, -1. Thus $\mathcal{P}$ has no Γ' QFC solution.

6.1 Computing Control Abstractions

Function minCtrAbs in Alg. 1 computes a *close to minimum* Γ control abstraction (Def. 8) $\hat{\mathcal{Q}} = (\Gamma(\mathcal{D}_X), \Gamma(\mathcal{D}_U), \hat{N})$ of $\mathcal{H} = (X, U, Y, N)$ as well as $\hat{I} = \Gamma(I)$ and $\hat{G} = \Gamma(G)$.

Line 6 initializes (the OBDDs for) $\hat{N}$, $\hat{I}$, $\hat{G}$ to $\varnothing$ (i.e. the boolean function identically 0). Line 2 loops through all $|\Gamma(\mathcal{D}_X)|$ states $\hat{s}$ of $\hat{\mathcal{H}}$. Line 3 [line 4] add state $\hat{s}$ to $\hat{I}$ [$\hat{G}$] if $\hat{s}$ is the image of a concrete state in I [G]. Line 5 loops through all $|\Gamma(\mathcal{D}_U)|$ actions $\hat{u}$ of $\hat{\mathcal{H}}$. Line 13 checks if action $\hat{u}$ is Γ-safe in $\hat{s}$ (see Def. 7[2] and Def. 8[2]). Function SelfLoop in line 7 returns 0 when, accordingly to Def. 8[3] a self-loop need not to be in $\hat{N}$. An exact check is undecidable (Remark 2), however our *gradient based* SelfLoop function typically turns out (Tab. 1 in Sect. 7) to be a quite tight overapproximation of the sets of (strictly needed) self-loops. We compute SelfLoop$(\hat{s}, \hat{u})$ as follows. For each real valued state component x_i, let $w_i[W_i]$= (min[max], $x_i' - x_i, N(X, U, Y, X') \wedge \Gamma(X) = \hat{s} \wedge \Gamma(U) = \hat{u}$). If for some i [$w_i \neq 0 \wedge W_i \neq 0 \wedge$ (w_i and W_i have the same sign)] then SelfLoop returns 0 (since any *long enough* sequence of concrete actions in $\Gamma^{-1}(\hat{u})$ will drive state component x_i outside of $\Gamma^{-1}(\hat{s})$), otherwise SelfLoop returns 1. Lines 9, 10, 11 compute a *quite tight* overapproximation (Over_Img) of the set of states reachable in one step from $\hat{s}$. Line 12 loops on all |Over_Img| abstract next states $\hat{s}'$ that may be reachable with the abstract outgoing transition $(\hat{s}, \hat{u})$ under consideration. Line 13 checks if there exists a concrete transition realizing the abstract transition $(\hat{s}, \hat{u}, \hat{s}')$ when $\hat{s} \neq \hat{s}'$ (no self-loop) and if so adds the abstract transition $(\hat{s}, \hat{u}, \hat{s}')$ to $\hat{N}$ (line 14). Finally, line 15 returns the (transition relation for) the control abstraction along with $\hat{I}$ and $\hat{G}$.

Remark 3. From the loops in lines 2, 5, 12 we see that the worst case runtime for Alg. 1 is $\mathcal{O}(|\Gamma(\mathcal{D}_X)|^2|\Gamma(\mathcal{D}_U)|)$. However, thanks to the heuristic in lines 9–11, Alg. 1 typical runtime is about $\mathcal{O}(|\Gamma(\mathcal{D}_X)||\Gamma(\mathcal{D}_U)|)$ as confirmed by our experimental results (Sect. 7, Fig. 3(b)).

Remark 4. Alg. 1 is explicit in the (abstract) states and actions of $\hat{\mathcal{H}}$ and symbolic with respect to the auxiliary variables (*modes*) in the transition relation N of $\mathcal{H}$. As a result our approach will work well with systems with just a few state variables and many modes, our target here.

6.2 Control Software with a Guaranteed WCET

From controller $\hat{K}$ computed by QKS (see Sect. 6) we generate our correct-by-construction control software (obdd2c($\hat{K}$)). This is done (function obdd2c) by translating the OBDD representing $\hat{K}$ into C code along the lines of [29]. From such a construction we can easily compute the *Worst Case Execution Time* (WCET) for our controller. We have: $WCET = nrT_B$, where r [n] is the number of bits used to represent plant actions [states] and T_B is the time needed to execute the C instructions modelling the `if-then-else` semantics of OBDD nodes as well as edge complementation (since we use the CUDD package).

Algorithm 1. Building control abstractions

Input: A quantization Γ, a DTLHS $\mathcal{H} = (X, U, Y, N)$, a control problem $(\mathcal{H}, I, G)$.
function minCtrAbs(Γ, $\mathcal{H}$, I, G):

1. $\hat{N} \leftarrow \varnothing, \hat{I} \leftarrow \varnothing, \hat{G} \leftarrow \varnothing$, **let** $X = [x_1, \ldots, x_n]$, $X' = [x'_1, \ldots, x'_n]$
2. **for all** $\hat{s} \in \Gamma(\mathcal{D}_X)$ **do**
3. **if** (MILP (min, 0, $I(X) \wedge \Gamma(X) = \hat{s}$) is feasible) **then** $\hat{I} \leftarrow \hat{I} \cup \{\hat{s}\}$
4. **if** (MILP (min, 0, $G(X) \wedge \Gamma(X) = \hat{s}$) is feasible) **then** $\hat{G} \leftarrow \hat{G} \cup \{\hat{s}\}$
5. **for all** $\hat{u} \in \Gamma(\mathcal{D}_U)$ **do**
6. **if** (MILP (min, 0, $N(X, U, Y, X') \wedge \Gamma(X) = \hat{s} \wedge \Gamma(U) = \hat{u} \wedge X' \notin \mathcal{D}_X$) is feasible) **then continue**
7. **if** SelfLoop($\hat{s}$, $\hat{u}$) **then** $\hat{N} \leftarrow \hat{N} \cup \{(\hat{s}, \hat{u}, \hat{s})\}$
8. **for all** $i = 1, \ldots n$ **do**
9. $m_i \leftarrow x'^{*}_i$, where $X'^{*} = [x'^{*}_1, \ldots, x'^{*}_n]$ is a solution to the MILP (min, x'_i, $N(X, U, Y, X') \wedge \Gamma(X) = \hat{s} \wedge \Gamma(U) = \hat{u}$)
10. $M_i \leftarrow x'^{*}_i$, where $X'^{*} = [x'^{*}_1, \ldots, x'^{*}_n]$ is a solution to the MILP (max, x'_i, $N(X, U, Y, X') \wedge \Gamma(X) = \hat{s} \wedge \Gamma(U) = \hat{u}$)
11. **let** Over_Img($\hat{s}, \hat{u}$) $= \prod_{i=1,\ldots n} [\gamma_{x_i}(m_i), \gamma_{x_i}(M_i)]$
12. **for all** $\hat{s}' \in$ Over_Img($\hat{s}, \hat{u}$) **do**
13. **if** $\hat{s} \neq \hat{s}' \wedge$ (MILP (min, 0, $N(X, U, Y, X') \wedge \Gamma(X) = \hat{s} \wedge \Gamma(U) = \hat{u} \wedge \Gamma(X') = \hat{s}'$) is feasible) **then**
14. $\hat{N} \leftarrow \hat{N} \cup \{(\hat{s}, \hat{u}, \hat{s}')\}$
15. **return** $(\hat{N}, \hat{I}, \hat{G})$

Let T be the chosen sampling time. Then it must be: $WCET \leq T$. That is, $nrT_B \leq T$. This equation allows us to know, before hand, the realizability (e.g. with respect to schedulability constraints) of the (to be designed) control software. For example, let $T_B = 10^{-7}$secs, $n = 10$ and $r = 1$. Then, the for the system sampling time we have: $T \geq 10^{-6} = WCET$.

7 Experimental Results

We implemented QKS (Sect. 6) in C, using GLPK to solve MILP problems and the CUDD package for OBDD based computations.

Our experiments aim at evaluating effectiveness of: control abstraction ($\hat{\mathcal{Q}}$, Sect. 6.1) generation, synthesis of OBDD representation of control law ($\hat{K}$, Sect. 6), control software (obdd2c($\hat{K}$), Sect. 6.2) size and guaranteed operational ranges (i.e. controllable region). Note that control software reaction time (WCET) is known a priori from Sect. 6.2 and its robustness to parameter variations in the controlled system ($\mathcal{H}$) as well as enforcement of safety bounds on state variables are an input to our synthesis algorithm (e.g. see Ex. 1, 2).

We present experimental results obtained by using QKS on the buck DC-DC converter described in Ex. 2. We denote with $\mathcal{H}$ the DTLHS modeling such a converter. We set the parameters of $\mathcal{H}$ as follows: $T = 10^{-6}$ secs, $L = 2 \cdot 10^{-4}$ H, $r_L = 0.1\ \Omega$, $r_C = 0.1\ \Omega$, $R = 5 \pm 25\%\ \Omega$, $C = 5 \cdot 10^{-5}$ F, $V_i = 15 \pm 25\%$ V and require our controller to be robust to foreseen variations (25%) in the load (R) and in the power supply (V_i).

The model in Ex. 2 already accounts for variations in the power supply. Variations in the load R can be taken into account along the same lines, however much more work is needed (along the lines of [15]) since $\mathcal{H}$ dynamics is not linear in R. This adds 11 auxiliary boolean variables to the model in Ex. 2. Details are in [23]. For converters, *safety* (as well as physical) considerations set requirements on admissible values for state variables. We set: $\mathcal{D}_{i_L} = [-4, 4]$, $\mathcal{D}_{v_O} = [-1, 7]$. Note that robustness requires that, notwithstanding nondeterministic variations (within the given tolerances) for power supply and load, the synthesized controller always keeps state variables within their admissible regions. We use the following bounds for auxiliary variables: $\mathcal{D}_{i_u} = \mathcal{D}_{i_D} = [-10^3, 10^3]$ and $\mathcal{D}_{v_u} = \mathcal{D}_{v_D} = [-10^7, 10^7]$. The initial region I and goal region G are as in Ex. 5. Finally, the DTLHS control problem we consider is $P = (\mathcal{H}, I, G)$. Note that no (formally proved) robust control software is available for buck DC-DC converters.

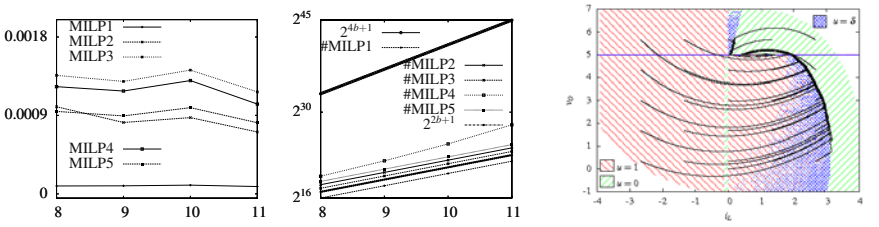
We use a uniform quantization dividing the domain of each state variable (i_L, v_O) into 2^b equal intervals, where b is the number of bits used by AD conversion. We call the resulting quantization Γ_b. The *quantization step* is $\|\Gamma_b\| = 2^{3-b}$.

For each value of interest for b, following Sect. 6, we compute: (1) a (close to minimum) Γ_b control abstraction $\hat{\mathcal{H}}^b$ for $\mathcal{H}$, (2) the mgo strong solution $\hat{K}^b$ for $\hat{\mathcal{P}}^b = (\hat{\mathcal{H}}^b, \varnothing, \Gamma_b(G))$, (3) $\hat{K}^b$ controllable region $\hat{D}^b = \mathrm{Dom}(\hat{K}^b)$, (4) a Γ_b QFC solution $K^b(s, u) = \hat{K}^b(\Gamma_b(s), \Gamma_b(u))$ to the control problem $\mathcal{P}^b = (\mathcal{H}, \Gamma_b^{-1}(\hat{D}^b), G)$. Note that, since we have two quantized variables (i_L, v_O) each one with b bits, the number of states in the control abstraction is exactly 2^{2b}.

Tab. 1 shows our experimental results. Columns in Tab. 1 have the following meaning. Column b shows the number of AD bits. Columns labelled *Control Abstraction* show performances for Alg. 1. Column *CPU* shows Alg. 1 time (in secs) to compute $\hat{\mathcal{H}}^b$. Column *Arcs* shows the number of transitions in $\hat{\mathcal{H}}^b$. In order to assess effectiveness of function SelfLoop (Sect. 6.1) column *MaxLoops* shows the number of loops in the maximum Γ_b control abstraction for $\mathcal{H}$, while column *LoopFrac* shows the fraction of such loops in $\hat{\mathcal{H}}^b$. Columns labelled *Controller Synthesis* show the computation time in secs (*CPU*) for the generation of $\hat{K}^b$, and the size of its OBDD representation (*OBDD*). The latter is also the size (number of lines) of the C code for our synthesized implementation of $\hat{K}^b$

Table 1. Buck DC-DC converter (Sect. 3): control abstraction and controller synthesis results. Experiments run on an Intel 3.0 GHz Dual Core Linux PC with 4 GB of RAM.

	Control Abstraction				Controller Synthesis		Total
b	CPU	Arcs	MaxLoops	LoopFrac	CPU	OBDD	CPU
8	2.50e+03	1.35e+06	2.54e+04	0.00323	0.00e+00	1.07e+02	2.50e+03
9	1.13e+04	7.72e+06	1.87e+04	0.00440	1.00e+02	1.24e+03	1.14e+04
10	6.94e+04	5.14e+07	2.09e+04	0.00781	7.00e+02	2.75e+03	7.01e+04
11	4.08e+05	4.24e+08	2.29e+04	0.01417	5.00e+03	7.00e+03	4.13e+05



(a) Average execution time for MILP problems in Alg.1

(b) Number of calls to MILP problems in Alg.1

(c) Controllable region with $b = 10$ bits (δ stands for "don't care")

Fig. 3. QKS performance

($\mathsf{obdd2c}(\hat{K}^b)$). Finally, column *Total* shows the total computation time in secs (*CPU*) for the whole process (i.e., control abstraction plus controller source code generation). All computations were completed using no more than 200MB. As for the value of μ (see Theor. 1), we have that $\mu = \textsc{Unk}$ for $b = 8$, and $\mu = \textsc{Sol}$ in all other cases.

From Tab. 1 we see that computing control abstractions (i.e. Alg. 1) is the most expensive operation in QKS (see Sect. 6) and that thanks to function $\mathsf{SelfLoop}$ $\hat{K}^b$ contains no more than 2% of the loops in the maximum Γ_b control abstraction for $\mathcal{H}$.

For each MILP problem in Alg. 1, Fig. 3(b) shows (as a function of b) the number of MILP instances solved while Fig. 3(a) shows (as a function of b) the average CPU time (in seconds) spent solving a single MILP problem instance. CPU time standard deviation is always less than 0.003. The correspondence between the curves in Figs. 3(b), 3(a) and Alg. 1 is the following. MILP1 refers to line 3 (and represents also the data for the twin MILP in line 4). MILP2 refers to MILP problems in function $\mathsf{SelfLoop}$ (line 7). MILP3 refers to line 9 (and represents also the data for the twin MILP in line 10). MILP4 refers to line 13 and MILP5 refers to line 6.

From Fig.3(a) we see that the average time spent solving each MILP instance is small. The lower [upper] bound to the number of times MILP4 (i.e. the most called MILP in Alg. 1) is called (#MILP4) is $|\Gamma(\mathcal{D}_X)||\Gamma(\mathcal{D}_U)| = 2^{2b+1}$ $[|\Gamma(\mathcal{D}_X)|^2|\Gamma(\mathcal{D}_U)| = 2^{4b+1}]$ (see Remark 3). From Fig. 3(b) we see that #MILP4 is quite close to $|\Gamma(\mathcal{D}_X)||\Gamma(\mathcal{D}_U)| = 2^{2b+1}$. This shows effectiveness of our heuristic to tightly overapproximate $\mathsf{Over\_Img}$ (lines 9–11 of Alg. 1).

One of the most important features of our approach is that it returns the guaranteed operational range (precondition) of the synthesized software (Theor. 1). This is the *controllable region* D returned by QKS in Sect. 6. Fig. 3(c) shows the controllable region D for K^{10} along with some trajectories (with time increasing counterclockwise) for the closed loop system. Since for $b = 10$ we have $\mu = \textsc{Sol}$, we have that $I \subseteq D$ (see also Fig. 3(c)). Thus we know (on a formal ground) that 10 bit AD ($\|\Gamma_{10}\| = 2^{-7}$) conversion suffices for our purposes. The controllable region for K^{11} turns out to be only slightly larger than the one for K^{10}.

8 Conclusion

We presented an effective algorithm that given a DTLHS $\mathcal{H}$ and a quantization schema returns a correct-by-construction robust control software K for $\mathcal{H}$ along with the controllable region R for K. Furthermore, our control software has a WCET linear in the number of bits of the quantization schema. We have implemented our algorithm and shown feasibility of our approach by presenting experimental results on using it to synthesize C controllers for the buck DC-DC converter. Our approach is explicit in the quantized state variables and symbolic in the system modes. Accordingly, it works well with systems with a small number of (continuous) state variables and possibly many modes. Many hybrid systems fall in this category.

Future research may investigate fully symbolic approaches, e.g., based on Fourier-Motzkin (FM) variable elimination, to compute control abstractions. Since FM tools typically work on rational numbers this would also have the effect of avoiding possible numerical errors of MILP solvers [24].

Acknowledgments. We are grateful to our anonymous referees for their helpful comments. Our work has been partially supported by: MIUR project DM24283 (TRAMP) and by the EC FP7 project GA218815 (ULISSE).

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138(1), 3–34 (1995)
2. Alur, R., Dang, T., Ivančić, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Trans. on Embedded Computing Sys. 5(1), 152–199 (2006)
3. Alur, R., Henzinger, T.A., Ho, P.-H.: Automatic symbolic verification of embedded systems. IEEE Trans. Softw. Eng. 22(3), 181–201 (1996)
4. Alur, R., Madhusudan, P.: Decision problems for timed automata: A survey. In: SFM, pp. 1–24 (2004)
5. Asarin, E., Bournez, O., Dang, T., Maler, O., Pnueli, A.: Effective synthesis of switching controllers for linear systems. Proc. of the IEEE 88(7), 1011–1025 (2000)
6. Asarin, E., Maler, O.: As soon as possible: Time optimal control for timed automata. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 19–30. Springer, Heidelberg (1999)
7. Attie, P.C., Arora, A., Emerson, E.A.: Synthesis of fault-tolerant concurrent programs. ACM Trans. on Program. Lang. Syst. 26(1), 125–185 (2004)
8. Bemporad, A., Giorgetti, N.: A sat-based hybrid solver for optimal control of hybrid systems. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 126–141. Springer, Heidelberg (2004)
9. Bouyer, P., Brihaye, T., Chevalier, F.: O-minimal hybrid reachability games. Logical Methods in Computer Science 6(1:1) (January 2010)
10. Bryant, R.: Graph-based algorithms for boolean function manipulation. IEEE Trans. on Computers C-35(8), 677–691 (1986)
11. Cassez, F., David, A., Fleury, E., Larsen, K.G., Lime, D.: Efficient on-the-fly algorithms for the analysis of timed games. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 66–80. Springer, Heidelberg (2005)

12. Cimatti, A., Roveri, M., Traverso, P.: Strong planning in non-deterministic domains via model checking. In: AIPS, pp. 36–43 (1998)
13. Dominguez-Garcia, A., Krein, P.: Integrating reliability into the design of fault-tolerant power electronics systems. In: PESC, pp. 2665–2671. IEEE, Los Alamitos (2008)
14. Fu, M., Xie, L.: The sector bound approach to quantized feedback control. IEEE Trans. on Automatic Control 50(11), 1698–1711 (2005)
15. Henzinger, T.A., Horowitz, B., Majumdar, R., Wong-Toi, H.: Beyond hytech: Hybrid systems analysis using interval numerical methods. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 130–144. Springer, Heidelberg (2000)
16. Henzinger, T.A., Kopke, P.W.: Discrete-time control for rectangular hybrid automata. In: Degano, P., Gorrieri, R., Marchetti-Spaccamela, A. (eds.) ICALP 1997. LNCS, vol. 1256, pp. 582–593. Springer, Heidelberg (1997)
17. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata? J. of Computer and System Sciences 57(1), 94–124 (1998)
18. 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)
19. Jha, S., Brady, B.A., Seshia, S.A.: Symbolic reachability analysis of lazy linear hybrid automata. In: Raskin, J.-F., Thiagarajan, P.S. (eds.) FORMATS 2007. LNCS, vol. 4763, pp. 241–256. Springer, Heidelberg (2007)
20. 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)
21. Liu, X., Ding, H., Lee, K., Wang, Q., Sha, L.: Ortega: An efficient and flexible software fault tolerance architecture for real-time control systems. IEEE Trans. On: Industrial Informatics 4(4) (November 2008)
22. Maler, O., Nickovic, D., Pnueli, A.: On synthesizing controllers from bounded-response properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 95–107. Springer, Heidelberg (2007)
23. Mari, F., Melatti, I., Salvo, I., Tronci, E.: Synthesis of quantized feedback control software for discrete time linear hybrid systems. Technical report, Computer Science Department, La Sapienza University of Rome (January 2010)
24. Neumaier, A., Shcherbina, O.: Safe bounds in linear and mixed-integer programming. Mathematical Programming, Ser. A 99, 283–296 (2004)
25. Olivero, A., Sifakis, J., Yovine, S.: Using abstractions for the verification of linear hybrid systems. In: Dill, D.L. (ed.) CAV 1994. LNCS, vol. 818, pp. 81–94. Springer, Heidelberg (1994)
26. So, W.-C., Tse, C., Lee, Y.-S.: Development of a fuzzy logic controller for dc/dc converters: design, computer simulation, and experimental evaluation. IEEE Trans. on Power Electronics 11(1), 24–32 (1996)
27. Tabuada, P., Pappas, G.J.: Linear time logic control of linear systems. IEEE Trans. on Automatic Control (2004)
28. Tomlin, C., Lygeros, J., Sastry, S.: Computing controllers for nonlinear hybrid systems. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 238–255. Springer, Heidelberg (1999)
29. Tronci, E.: Automatic synthesis of controllers from formal specifications. In: ICFEM, p. 134. IEEE Computer Society, Los Alamitos (1998)
30. Wong-Toi, H.: The synthesis of controllers for linear hybrid automata. In: CDC, vol. 5, pp. 4607–4612 (1997)

Safety Verification for Probabilistic Hybrid Systems

Lijun Zhang[1], Zhikun She[2], Stefan Ratschan[3],
Holger Hermanns[4], and Ernst Moritz Hahn[4]

[1] DTU Informatics, Technical University of Denmark, Denmark
[2] LMIB and School of Mathematics and Systems Science, Beihang University, China
[3] Institute of Computer Science, Czech Academy of Sciences, Czech Republic
[4] Department of Computer Science, Saarland University, Germany

Abstract. The interplay of random phenomena and continuous real-time control deserves increased attention for instance in wireless sensing and control applications. Safety verification for such systems thus needs to consider probabilistic variations of systems with hybrid dynamics. In safety verification of classical hybrid systems we are interested in whether a certain set of unsafe system states can be reached from a set of initial states. In the probabilistic setting, we may ask instead whether the probability of reaching unsafe states is below some given threshold. In this paper, we consider probabilistic hybrid systems and develop a general abstraction technique for verifying probabilistic safety problems. This gives rise to the first mechanisable technique that can, in practice, formally verify safety properties of non-trivial continuous-time stochastic hybrid systems—without resorting to point-wise discretisation. Moreover, being based on arbitrary abstractions computed by tools for the analysis of non-probabilistic hybrid systems, improvements in effectivity of such tools directly carry over to improvements in effectivity of the technique we describe. We demonstrate the applicability of our approach on a number of case studies, tackled using a prototypical implementation.

1 Introduction

Conventional hybrid system formalisms [1–4] capture many characteristics of real systems (telecommunication networks, air traffic management, etc.). However, in some modern application areas, the lack of randomness hampers faithful modelling and verification. This is especially true for wireless sensing and control applications, where message loss probabilities and other random effects (node placement, node failure, battery drain) turn the overall control problem into a problem that can only be managed with a certain, hopefully sufficiently large, probability.

The idea of integrating probabilities into hybrid systems is not new, and different models have been proposed, each from its own perspective [5–9]. The most important difference lies in the place where to introduce randomness. One option is to replace deterministic jumps by probability distributions over deterministic jumps. Another option is to replace differential equations in each mode by stochastic differential equations. More general models can be obtained by blending the above two choices, and by combining with memoryless timed probabilistic jumps [10].

An important problem in hybrid systems theory is that of reachability analysis. In general terms, a reachability analysis problem consists in evaluating whether a given

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 196–211, 2010.

system will reach certain unsafe states, starting from certain initial states. This problem is associated with the safety verification problem: if the system cannot reach any unsafe state, then the system is declared to be safe. In the probabilistic setting, the safety verification problem can be formulated as that of checking whether the probability that the system trajectories reach an unsafe state from its initial states can be bounded by some given probability threshold.

In this paper, we focus on the probabilistic hybrid automata model [6], an extension of hybrid automata where the jumps involve probability distributions. This makes it possible to model component failures, message losses, buffer overflows and the like. Since these phenomena are important aspects when aiming at faithful models for networked and embedded applications, the interest in this formalism is growing [11,12].

Up to now, foundational results on the probabilistic reachability problem for probabilistic hybrid automata are scarce. Since they form a strict superclass of hybrid automata, this is not surprising. Decidability results are known for probabilistic linear hybrid automata and o-minimal hybrid automata [6].

This paper reports how we harvest and combine recent advances in the hybrid automata and the probabilistic automata worlds, in order to treat the general case. We are doing so by computing safe over-approximations via abstractions in the continuous as well as the probabilistic domain. One of the core challenges then is how to construct a sound probabilistic abstraction over a given covering of the state space. For this purpose, we first consider the non-probabilistic hybrid automaton obtained by replacing probabilistic branching with nondeterministic choices. Provided that there is a finite abstraction for this classical hybrid automaton, we then decorate this abstraction with probabilities to obtain a probabilistic abstraction, namely a finite probabilistic automaton [13]. We show the soundness of this abstraction, which allows us to verify probabilistic safety properties on the abstraction: if such a property holds in the abstraction, it holds also in the concrete system. Otherwise, refinement of the abstraction is required to obtain a more precise result.

Our abstraction approach can be considered as an orthogonal combination of the abstraction for hybrid automata [4,14], and Markov decision processes [15,16]. Because of this orthogonality, abstractions of probabilistic hybrid automata can be computed via abstractions for non-probabilistic hybrid automata and Markov decision processes. To show the applicability of this combination, we implemented a prototype tool, `ProHVer`, that first builds an abstraction via existing techniques [17] for classical hybrid automata, and then via techniques for Markov decision processes [15,16,18]. Subsequently, a fixed-point engine computes the reachability probabilities on the abstraction, which provides a safe upper bound. If needed, iterative refinement of the hybrid abstraction is performed. We report several successful applications of this prototypical implementation on different case studies. To the best of our knowledge, this is the first implementation which automatically checks safety properties for probabilistic hybrid automata.

2 Related Work

The verification of safety properties is undecidable for general hybrid automata. However, certain classes (e.g., initialised rectangular automata [19], o-minimal hybrid

automata [20]), are decidable, and there are algorithms that construct finite bisimulation quotient automata. These results have been lifted to probabilistic hybrid automata [6], and provide exact results, rooted in a bisimulation-based abstraction. In these special cases, our approach can yield the same results, but it gives us the freedom to use different abstractions, that are more adapted to the problem at hand, but then not exact, but over-approximating. We actually treat the general case using that a practical verification can —to a certain extent— circumvent the decidability barrier by a semi-decision algorithm: we exploit tools that can, in practice, verify hybrid automata belonging to undecidable classes, to verify corresponding probabilistic hybrid automata.

The abstraction approach has also successfully been applied to probabilistic timed automata [18,21], a class of probabilistic hybrid automata, where only derivatives of constant value 1 occur. Their abstract analysis is based on difference-bound matrices (DBMs), and does not extend to the general setting considered here. Fränzle et al. [11, 12] use stochastic SAT to solve reachability problems on probabilistic hybrid automata. Their analysis is limited to depth-bounded reachability properties, i.e., the probability of reaching a location within at most N discrete jumps.

While the model we consider has probabilistic discrete jumps, there are several other suggestions equipping hybrid automata with continuous-time jumps. Davis [22] introduced piecewise deterministic Markov processes, whose state changes are triggered spontaneously as in continuous-time Markov chains. Stochastic differential equations [23] incorporate the continuous dynamics with random perturbations, such as Brownian motion. In stochastic hybrid systems [24,25], the transitions between different locations are resolved via a race between different Poisson processes. While these models enjoy a variety of applications, their analysis are limited and often based on Monte-Carlo simulations [8,9,26,27].

3 Preliminaries

In this section, we repeat the definition of conventional hybrid automata, in the style of [4], followed by the definition of probabilistic hybrid automata [6].

3.1 Hybrid Automata

We fix a variable m ranging over a finite set of discrete modes $\mathtt{M} = \{m_1, \ldots, m_n\}$ and variables $x_1, \ldots, x_k$ ranging over reals $\mathbb{R}$. We denote by S the resulting state space $\mathtt{M} \times \mathbb{R}^k$. For denoting the derivatives of $x_1, \ldots, x_k$ we use variables $\dot{x}_1, \ldots, \dot{x}_k$, ranging over $\mathbb{R}$ correspondingly. For simplicity, we sometimes use the vector $\boldsymbol{x}$ to denote $(x_1, \ldots, x_k)$, and $(m, \boldsymbol{x})$ to denote a state. Similar notations are used for the primed and dotted versions $\boldsymbol{x'}$, $\dot{\boldsymbol{x}}$.

In order to describe hybrid automata we use constraints that are arbitrary Boolean combinations of equalities and inequalities over terms. These constraints are used, on the one hand, to describe the possible flows and jumps and, on the other hand, to mark certain parts of the state space (e.g., the set of initial/unsafe states). A *state space constraint* is a constraint over the variables $m, \boldsymbol{x}$. A *flow constraint* is a constraint over the variables m, $\boldsymbol{x}$, $\dot{\boldsymbol{x}}$.

For capturing the jump behaviours, we introduce the notion of update constraints. An *update constraint* $\mathtt{u}$, also called a *guarded command*, has the form: $condition \rightarrow update$ where $condition$ is a constraint over $m, \boldsymbol{x}$, and $update$ is an expression denoting a function $\mathtt{M} \times \mathbb{R}^k \rightarrow \mathtt{M} \times \mathbb{R}^k$ which is called the reset mapping for m and $\boldsymbol{x}$. Intuitively, assume that the state $(m, \boldsymbol{x})$ satisfies $condition$, then the mode m and variable $\boldsymbol{x}$ are updated[1] to the new state $update(m, \boldsymbol{x})$.

A *jump constraint* is a finite disjunction $\bigvee_{\mathtt{u} \in \mathtt{U}} \mathtt{u}$ where $\mathtt{U}$ is a set of guarded commands. The constraint $\bigvee_{\mathtt{u} \in \mathtt{U}} \mathtt{u}$ can be represented by the set $\mathtt{U}$ for simplicity.

A *hybrid automaton* is a tuple $\mathcal{H} = (Flow, \mathtt{U}, Init, UnSafe)$ consisting of a flow constraint $Flow$, a finite set of update constraints $\mathtt{U}$, a state space constraint $Init$ describing the set of initial states, and a state space constraint $UnSafe$ describing the set of unsafe states.

A *flow* of length l in a mode m is a function $r : [0, l] \mapsto \mathbb{R}^k$ with $l > 0$ such that r is differentiable for all $t \in [0, l]$, and for all $t \in [0, l]$, $(m, r(t), \dot{r}(t))$ satisfies $Flow$, where $\dot{r}$ is the derivative of r.

Transition System Semantics. The semantics of a hybrid automaton is a transition system with an uncountable set of states. Formally, the semantics of $\mathcal{H} = \{Flow, \mathtt{U}, Init, UnSafe\}$ is a transition system $T_{\mathcal{H}} = (S, T, S_{Init}, S_{UnSafe})$ where $S = \mathtt{M} \times \mathbb{R}^k$ is the set of states, $S_{Init} = \{s \in S \mid s \text{ satisfies } Init\}$ denotes the set of initial states, and $S_{UnSafe} = \{s \in S \mid s \text{ satisfies } UnSafe\}$ represents the set of unsafe states. The transition set T is defined as the union of two transition relations $T_{\mathcal{C}}, T_{\mathcal{D}} \subseteq S \times S$, where $T_{\mathcal{C}}$ corresponds to transitions due to continuous flows defined by:

- $((m, \boldsymbol{x}), (m, \boldsymbol{x'})) \in T_{\mathcal{C}}$, if there exists a flow r of length l in m such that $r(0) = \boldsymbol{x}$ and $r(l) = \boldsymbol{x'}$;

and $T_{\mathcal{D}}$ corresponds to transitions due to discrete jumps. The transition due to an update constraint $\mathtt{u} : condition \rightarrow update$, denoted by $T_{\mathcal{D}}(\mathtt{u})$ is defined by:

- $((m, \boldsymbol{x}), (m', \boldsymbol{x'})) \in T_{\mathcal{D}}(\mathtt{u})$ if $(m, \boldsymbol{x})$ satisfies the guard $condition$ and it holds that $(m', \boldsymbol{x'}) = update(m, \boldsymbol{x})$.

Then, we define $T_{\mathcal{D}} = \cup_{\mathtt{u} \in \mathtt{U}} T_{\mathcal{D}}(\mathtt{u})$.

In the rest of the paper, if no confusion arises, we use $Init$ to denote both the constraint for the initial states and the set of initial states. Similarly, $UnSafe$ is used to denote both the constraint for the unsafe states and the set of unsafe states.

3.2 Probabilistic Automata

For defining the semantics of a probabilistic hybrid automaton, we recall first the notion of a probabilistic automaton [13]. It is an extension of a transition system with probabilistic branching.

[1] Our definition of jumps is deterministic, as in [14], i.e., if a jump is triggered for a state satisfying $condition$, the successor state is updated deterministically according to $update$. In [4], the jump is defined to be nondeterministic: if a state satisfies $condition$, a successor will be selected nondeterministically from a set of states. Our method can be easily extended to this. We restrict to deterministic jumps for simplicity of the presentation in this paper.

We first introduce some notation. Let S be a (possibly uncountable) set. A *distribution* over S is a function $\mu : S \to [0,1]$ such that (a) the set $\{s \in S \mid \mu(s) > 0\}$ is finite, and (b) the sum $\sum_{s \in S} \mu(s) = 1$. Let the support $Supp(\mu)$ of μ be $\{s \in S \mid \mu(s) > 0\}$. Let $Distr(S)$ denote the set of all distributions over S. For an arbitrary but fixed state s in S, a *Dirac distribution* for s, denoted by $Dirac_s$, is a distribution over S such that $Dirac_s(s) = 1$, that is, $Supp(Dirac_s) = \{s\}$. Note that the Dirac distribution will be used to describe the continuous evolution of a probabilistic hybrid automaton.

Definition 1. *A probabilistic automaton $\mathcal{M}$ is a tuple $(S, Steps, Init, UnSafe)$, where $Steps \subseteq S \times Distr(S)$, $Init \subseteq S$, and $UnSafe \subseteq S$. Here, S denotes the (possible uncountable) set of states, $Init$ is the set of initial states, $UnSafe$ the set of unsafe states, and $Steps \subseteq S \times Distr(S)$ the transition relation.*

For a transition $(s, \mu) \in Steps$, we use $s \to \mu$ as a shorthand notation, and call μ a successor distribution of s. Let $Steps(s)$ be the set $\{\mu \mid (s, \mu) \in Steps\}$. We assume that $Steps(s) \neq \emptyset$ for all $s \in S$.

A path of $\mathcal{M}$ is a finite or infinite sequence $\sigma = s_0 \mu_0 s_1 \mu_1 \ldots$ such that $s_i \to \mu_i$ and $\mu_i(s_{i+1}) > 0$ for all possible $i \geq 0$. We denote by $first(\sigma)$ the first state s_0 of σ, by $\sigma[i]$ the $i+1$-th state s_i, and, if σ is finite, by $last(\sigma)$ the last state of σ. Let $Path$ be the set of all infinite paths and $Path^*$ the set of all finite paths.

The non-deterministic choices in $\mathcal{M}$ can be resolved by *adversaries*. Formally, an adversary of $\mathcal{M}$ is a map $A : Path^* \to Distr(Steps)$ such that $A(\sigma)(s, \mu) > 0$ implies that $s = last(\sigma)$ and $s \to \mu$. Intuitively, if $A(\sigma)(s, \mu) > 0$, then the successor distribution μ should be selected from state s with probability $A(\sigma)(s, \mu)$. Moreover, an adversary A is called *Markovian* if for all $\sigma \in Path^*$, $A(\sigma) = A(last(\sigma))$, that is, for each finite path, A depends only on its last state. An adversary A is called *deterministic* if for all $\sigma \in Path^*$, $A(\sigma)$ is always a Dirac distribution. We say that an adversary A is *simple* if A is Markovian and deterministic. Given an adversary A and an initial state s, a unique probability measure over $Path$, which is denoted by $Prob_s^A$, can be defined.

3.3 Probabilistic Hybrid Automata

Now we recall the definition of probabilistic hybrid automata, by equipping the discrete jumps with probabilities. This is needed to model, for example, component failure or message losses.

For capturing the probabilistic jump behaviours, a *guarded command* c is defined to have the form

$$condition \to p_1 : update_1 + \ldots + p_{q_\mathsf{c}} : update_{q_\mathsf{c}}$$

where $q_\mathsf{c} \geq 1$ denotes the number of probabilistic branching of c, $p_i > 0$ for $i = 1, \ldots, q_\mathsf{c}$ and $\sum_{i=1}^{q_\mathsf{c}} p_i = 1$, $condition$ is a constraint over $(m, \boldsymbol{x})$, and $update_i$ is an expression denoting a reset mapping for m and $\boldsymbol{x}$ for all $i = 1, \ldots, q_\mathsf{c}$. Intuitively, if a state $(m, \boldsymbol{x})$ satisfies the guard $condition$, a jump to states $(m_1, \boldsymbol{x}_1), \ldots, (m_{q_\mathsf{c}}, \boldsymbol{x}_{q_\mathsf{c}})$ occurs such that $(m_i, \boldsymbol{x}_i) = update_i(m, \boldsymbol{x})$ is selected with probability p_i for $i = 1, \ldots, q_\mathsf{c}$. Observe that for different $i \neq j$, it could be the case that $(m_i, \boldsymbol{x_i}) = (m_j, \boldsymbol{x_j})$. In this paper we assume that q_c is finite for all c.

Definition 2. *A* probabilistic hybrid automaton *is a tuple* $\mathcal{H} = (Flow, \mathsf{C}, Init, UnSafe)$ *where* $Flow$, $Init$, $UnSafe$ *are the same as in the hybrid automaton, and* C *is a finite set of guarded commands* C.

The probabilistic hybrid automaton induces a classical hybrid automaton where probabilistic branching is replaced by nondeterministic choices. Intuitively, the semantics of the latter spans the semantics of the former.

Definition 3. *Let* $\mathsf{c} : condition \to p_1 : update_1 + \ldots + p_{q_\mathsf{c}} : update_{q_\mathsf{c}}$ *be a guarded command. It induces a set of* q *update constraints:* $ind(\mathsf{c}) = \{\mathsf{u}_1, \ldots, \mathsf{u}_{q_\mathsf{c}}\}$ *where* u_i *corresponds to the update constraint* $condition \to update_i$ *for* $i = 1, \ldots, q_\mathsf{c}$. *Moreover, we define* $ind(\mathsf{C}) := \bigcup_{\mathsf{c}\in\mathsf{C}} ind(\mathsf{c})$.

Let $\mathcal{H} = (Flow, \mathsf{C}, Init, UnSafe)$ *be a probabilistic hybrid automaton. The induced hybrid automaton is a tuple* $ind(\mathcal{H}) = (Flow, ind(\mathsf{C}), Init, UnSafe)$.

Semantics. The semantics of a probabilistic hybrid automaton is a probabilistic automaton [6]. Let $\mathcal{H} = (Flow, \mathsf{C}, Init, UnSafe)$ be a probabilistic hybrid automaton. Let $ind(\mathcal{H})$ denote the induced hybrid automaton, and let $T_{ind(\mathcal{H})} = (S, T, Init, UnSafe)$ denote the transition system representing the semantics of $ind(\mathcal{H})$. Recall that $T = T_\mathcal{C} \cup T_\mathcal{D}$ where $T_\mathcal{C}$ corresponds to transitions due to continuous flow and $T_\mathcal{D}$ corresponds to transitions due to discrete jumps.

The semantics of $\mathcal{H}$ is the probabilistic automaton $\mathcal{M}_\mathcal{H} = (S, Steps, Init, UnSafe)$ where S, $Init$, $UnSafe$ are the same as in $T_{ind(\mathcal{H})}$, and $Steps$ is defined as the union of two transition relations $Steps_\mathcal{C}, Steps_\mathcal{D} \subseteq S \times Distr(S)$. Here, as in the non-probabilistic setting, $Steps_\mathcal{C}$ corresponds to transitions due to continuous flows, while $Steps_\mathcal{D}$ corresponds to transitions due to discrete jumps. Both of them are defined respectively as follows.

For each transition $((m, \boldsymbol{x}), (m, \boldsymbol{x'})) \in T_\mathcal{C}$ in $ind(\mathcal{H})$, there is a corresponding transition in $\mathcal{H}$ from $(m, \boldsymbol{x})$ to $(m, \boldsymbol{x'})$ with probability 1. So, $Steps_\mathcal{C}$ is defined by: $Steps_\mathcal{C} = \{((m, \boldsymbol{x}), Dirac_{(m,\boldsymbol{x'})}) \mid ((m, \boldsymbol{x}), (m, \boldsymbol{x'})) \in T_\mathcal{C}\}$.

Now we discuss transitions induced by discrete jumps. First, for a guarded command c, we define the set $Steps_\mathcal{D}(\mathsf{c})$ corresponding to it. Let $ind(\mathsf{c}) = \{\mathsf{u}_1, \ldots, \mathsf{u}_{q_\mathsf{c}}\}$ be as defined in Definition 3. Then, for arbitrary $q_\mathsf{c} + 1$ states $(m, \boldsymbol{x})$, $(m_1, \boldsymbol{x}_1), \ldots, (m_{q_\mathsf{c}}, \boldsymbol{x}_{q_\mathsf{c}}) \in S$ satisfying the condition $((m, \boldsymbol{x}), (m_i, \boldsymbol{x}_i)) \in T_\mathcal{D}(\mathsf{u}_i)$ for $i = 1, \ldots, q_\mathsf{c}$, we introduce the transition $((m, \boldsymbol{x}), \mu) \in Steps_\mathcal{D}(\mathsf{c})$ with

$$\mu(m_i, \boldsymbol{x}_i) = \sum_{j\in\{j \mid m_j = m_i \wedge \boldsymbol{x}_j = \boldsymbol{x}_i\}} p_j, \tag{1}$$

for $i = 1, \ldots, q_\mathsf{c}$. Then, $Steps_\mathcal{D}$ is defined to be $\bigcup_{\mathsf{c}\in\mathsf{C}} Steps_\mathcal{D}(\mathsf{c})$. Recall that we have assumed that q_c is finite for all c. This implies $Supp(\mu)$ is finite for all transitions (s, μ) with $s \in S$.

Safety Properties. For hybrid automata, the safety property asserts that the unsafe states can never be reached. For probabilistic hybrid automata, however, the safety property expresses that the maximal probability of reaching the set $UnSafe$ is bounded by some give threshold ε. In the following we fix a certain threshold ε. Let $Reach(UnSafe)$ denote the set of paths $\{\sigma \in Path \mid \exists i.\, \sigma[i] \in UnSafe\}$. The automaton $\mathcal{H}$ is called *safe*

if for each adversary A and each initial state s of $\mathcal{M}(\mathcal{H})$, $Prob_s^A(Reach(UnSafe)) \leq \varepsilon$ holds. In this paper, we would like to develop a framework to deal with such a probabilistic safety verification problem for general probabilistic hybrid automata.

Simulation Relations. We recall the notion of simulations between probabilistic automata. Intuitively, if $\mathcal{M}_2$ simulates $\mathcal{M}_1$, that is, $\mathcal{M}_2$ is an over-approximation of $\mathcal{M}_1$, then $\mathcal{M}_2$ can mimic all behaviours of $\mathcal{M}_1$. Thus, this allows us to verify safety properties on the abstraction $\mathcal{M}_2$ instead of $\mathcal{M}_1$. To establish the notion of simulations, we introduce first the notion of weight functions [28], which establish the correspondence between distributions.

Definition 4. *Let $\mu_1 \in Distr(S_1)$ and $\mu_2 \in Distr(S_2)$ be two distributions. For a relation $R \subseteq S_1 \times S_2$, a weight function for (μ_1, μ_2) with respect to R is a function $\Delta : S_1 \times S_2 \to [0,1]$ such that (i) $\Delta(s_1, s_2) > 0$ implies $(s_1, s_2) \in R$, (ii) $\mu_1(s_1) = \sum_{s_2 \in S_2} \Delta(s_1, s_2)$ for $s_1 \in S_1$, and (iii) $\mu_2(s_2) = \sum_{s_1 \in S_1} \Delta(s_1, s_2)$ for $s_2 \in S_2$.*

We write $\mu_1 \sqsubseteq_R \mu_2$ if and only if there exists a weight function for μ_1 and μ_2 with respect to R.

Now, we recall the notion of simulations [13]. The simulation requires that every successor distribution of a state of $\mathcal{M}_1$ is related to a successor distribution of its corresponding state of $\mathcal{M}_2$ via a weight function.

Definition 5. *Given two automata $\mathcal{M}_1 = (S_1, Init_1, Steps_1, UnSafe_1)$ and $\mathcal{M}_2 = (S_2, Init_2, Steps_2, UnSafe_2)$, we say that $\mathcal{M}_2$ simulates $\mathcal{M}_1$, denoted by $\mathcal{M}_1 \preceq \mathcal{M}_2$, if and only if there exists a relation $R \subseteq S_1 \times S_2$, which we will call simulation relation from now on, such that*

1. *for each $s_1 \in Init_1$ there exists an $s_2 \in Init_2$ with $(s_1, s_2) \in R$.*
2. *for each $s_1 \in UnSafe_1$ there exists an $s_2 \in UnSafe_2$ with $(s_1, s_2) \in R$.*
3. *for each pair $(s_1, s_2) \in R$, if there exists $(s_1, \mu_1) \in Steps_1$, there exists a distribution $\mu_2 \in Distr(S_2)$ such that $(s_2, \mu_2) \in Steps_2$ and $\mu_1 \sqsubseteq_R \mu_2$.*

4 Abstractions for Probabilistic Hybrid Automata

Various abstraction refinement techniques have been developed for verifying safety properties against non-probabilistic hybrid automata. All of them have a common strategy: the set S is covered by a finite set of abstract states, each representing a set of concrete states. Then, the abstraction is constructed which is an over-approximation of the original system. Afterwards, the safety property is checked on the abstraction. If the set of unsafe states is unreachable, the original system is safe since the abstraction over-approximates the original system. If not, the covering might have been chosen too coarse, and a refinement step is needed. Based on this idea, predicate abstraction based abstraction refinement has been used [3,14] for safety verification of linear hybrid automata, and constraint propagation based abstraction refinement has been used for safety verification of general hybrid automata [4].

Let $\mathcal{H} = (Flow, \subset, Init, UnSafe)$ be a probabilistic hybrid automaton. The aim of this section is—independent of which abstraction technique is used—to develop a framework for constructing an abstraction for $\mathcal{H}$, which is a finite probabilistic

automaton. First we introduce the notion of abstract states which form a (not necessarily disjoint) covering of the concrete state space:

Definition 6. *An abstract state is a pair* (m, B) *where* $m \in \mathcal{M}$ *and* $B \subseteq \mathbb{R}^k$. *The set* $\mathcal{B}$ *is a finite set of abstract states such that* $S = \bigcup\{(m, \boldsymbol{x}) \mid (m, B) \in \mathcal{B} \wedge \boldsymbol{x} \in B\}$.

In the above definition, any two abstract states (m, B_1) and (m, B_2) may have common interiors, including common borders[2]. The case allowing common interiors is the case if the polyhedra based abstraction technique is used [17], and common border is the case if the constraint propagation based abstraction technique is used [4]. Our abstraction scheme in this section works for all of them.

Fig. 1 illustrates how this section is organised. Given a probabilistic hybrid automaton $\mathcal{H}$ and an abstract state space $\mathcal{B}$, we introduce the quotient automaton for both $ind(\mathcal{H})$ and $\mathcal{H}$ in Sec. 4.1, respectively. In Sec. 4.2, we show the soundness with respect to the quotient automaton (cf. Lemma 1 and Lemma 2).

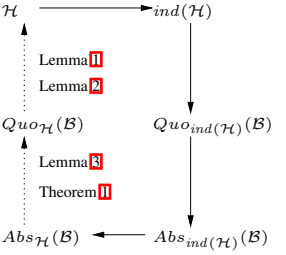


Fig. 1. Computation of the abstraction

The quotient automaton is in general hard to compute. Thus, we introduce in Sec. 4.3 general abstractions, which over-approximate the quotient automata conservatively. In Sec. 4.4, we discuss how the abstraction for the given probabilistic hybrid automaton is constructed (see Fig. 1): we construct first the abstraction of the induced hybrid automaton, from which the abstraction of the probabilistic setting is then obtained.

4.1 Quotient Automaton for $\mathcal{H}$

We define the quotient automaton for the probabilistic hybrid automaton $\mathcal{H}$. First we define the quotient automaton for the induced hybrid automaton $ind(\mathcal{H})$. As a convention we use $\mathcal{T}, \mathcal{I}, \mathcal{U}$ to denote the set of transitions, initial states, unsafe states in the quotient automata.

Definition 7. *Let* $\mathcal{H} = (Flow, C, Init, UnSafe)$ *be a probabilistic hybrid automaton, and let* $\mathcal{B}$ *denote the abstract state space. Let* $T_{ind(\mathcal{H})} = (S, T_C \cup T_D, Init, UnSafe)$ *denote the automaton representing the semantics of* $ind(\mathcal{H})$. *The quotient automaton for* $T_{ind(\mathcal{H})}$, *denoted by* $Quo_{ind(\mathcal{H})}(\mathcal{B})$, *is a finite transition system* $(\mathcal{B}, \mathcal{T}, \mathcal{I}, \mathcal{U})$ *where*

- $\mathcal{I} = \{(m, B) \in \mathcal{B} \mid \exists \boldsymbol{x} \in B.\ (m, \boldsymbol{x}) \in Init\}$,
- $\mathcal{U} = \{(m, B) \in \mathcal{B} \mid \exists \boldsymbol{x} \in B.\ (m, \boldsymbol{x}) \in UnSafe\}$,
- $\mathcal{T}_C$ *corresponds to the set of abstract transitions due to continuous flow:* $\mathcal{T}_C = \{((m, B), (m, B')) \in \mathcal{B}^2 \mid \exists \boldsymbol{x} \in B \wedge \exists \boldsymbol{x'} \in B' \wedge ((m, \boldsymbol{x}), (m, \boldsymbol{x'})) \in T_C\}$,
- $\mathcal{T}_D$ *corresponds to the set of abstract transitions due to discrete jumps. We first define the transition induced by one fixed update* $u \in ind(C)$. *Assume that we have*

[2] We may also require that abstract states form a partitioning over the original state S, with pairwise disjoint abstract states. Such abstractions are, however, harder to construct for non-trivial models.

$((m, \boldsymbol{x}), (m', \boldsymbol{x'})) \in Steps_{\mathcal{D}}(u)$. Then, it induces an abstract transition $((m, B), (m', B')) \in \mathcal{T}_{\mathcal{D}}(u)$ where B, B' are the abstract states containing $\boldsymbol{x}$, $\boldsymbol{x'}$ respectively. Then, let $\mathcal{T}_{\mathcal{D}} = \cup_{u \in ind(C)} \mathcal{T}_{\mathcal{D}}(u)$.

Let $\mathcal{H} = (Flow, \mathtt{C}, Init, UnSafe)$ be a probabilistic hybrid automaton, and let $\mathcal{M}_{\mathcal{H}} = (S, Steps_{\mathcal{C}} \cup Steps_{\mathcal{D}}, Init, UnSafe)$ denote the automaton representing the semantics of $\mathcal{H}$. As in the induced non-probabilistic setting, we define a quotient automaton, denoted by $Quo_{\mathcal{H}}(\mathcal{B})$, for an abstract state space $\mathcal{B}$. For this we first introduce the set of lifted distributions:

Definition 8. *Let $\mathcal{H}$ and $\mathcal{M}_{\mathcal{H}}$ be as described above. Let $\mathcal{B}$ denote the abstract state space. Let $\mathtt{c} \in \mathtt{C}$ and assume that $(s, \mu) \in Steps_{\mathcal{D}}(\mathtt{c})$ in $\mathcal{M}_{\mathcal{H}}$. By definition of $Steps_{\mathcal{D}}(\mathtt{c})$, there exist states $(m_1, \boldsymbol{x}_1), \ldots, (m_{q_{\mathtt{c}}}, \boldsymbol{x}_{q_{\mathtt{c}}}) \in S$ satisfying the condition $((m, \boldsymbol{x}), (m_i, \boldsymbol{x}_i)) \in T_{\mathcal{D}}(\mathtt{u}_i)$ for $i = 1, \ldots, q_{\mathtt{c}}$. Then, for arbitrary abstract states $(m_1, B_1), \ldots, (m_{q_{\mathtt{c}}}, B_{q_{\mathtt{c}}})$ with $\boldsymbol{x_i} \in B_i$ for $i = 1, \ldots, q_{\mathtt{c}}$ we introduce the distribution $\mu' \in Distr(\mathcal{B})$ by: $\mu'(m_i, B_i) = \sum_{\{j \mid (m_j, B_j) = (m_i, B_i)\}} \mu(m_j, \boldsymbol{x_j})$. The set of lifted distributions $lift_{\mathcal{B}}(\mu)$ contains all such μ'.*

Let μ be the distribution according to a guarded command $\mathtt{c}$. Since the covering $\mathcal{B}$ is in general not disjoint, a concrete state $(m_i, \boldsymbol{x_i})$ might belong to more than one abstract states. In this case μ induces more than one lifted distribution. In the above definition, this is reflected by the way of defining one specific lifted distribution μ', for which we first fix to which abstract state each concrete state $(m_i, \boldsymbol{x_i})$ belongs. Note that if $\mathcal{B}$ is a disjoint partitioning of S, the set $lift_{\mathcal{B}}(\mu)$ is a singleton. We now introduce the quotient automaton for the probabilistic hybrid automaton:

Definition 9. *Let $\mathcal{H}$ and $\mathcal{M}_{\mathcal{H}}$ be as described above. Let $\mathcal{B}$ denote the abstract state space. The quotient automaton for $\mathcal{M}_{\mathcal{H}}$ with respect to $\mathcal{B}$ is defined by $Quo_{\mathcal{H}}(\mathcal{B}) = (\mathcal{B}, \mathcal{ST}, \mathcal{I}, \mathcal{U})$ where $\mathcal{I}$ and $\mathcal{U}$ are defined as for $Quo_{ind(\mathcal{H})}(\mathcal{B})$, and $\mathcal{ST} = \mathcal{ST}_{\mathcal{C}} \cup \mathcal{ST}_{\mathcal{D}}$ is the set of abstract transitions where:*

- *$\mathcal{ST}_{\mathcal{C}}$ corresponds to the set of abstract transitions due to continuous flow: $\mathcal{ST}_{\mathcal{C}} = \{((m, B), Dirac_{(m, B')}) \mid \exists \boldsymbol{x} \in B \wedge \exists \boldsymbol{x'} \in B' \wedge ((m, \boldsymbol{x}), (m, \boldsymbol{x'})) \in Steps_{\mathcal{C}}\}$.*
- *$\mathcal{ST}_{\mathcal{D}}$ corresponds to the set of abstract transitions due to discrete jumps. We first define the transition induced by one fixed guarded command $\mathtt{c}$. Consider all $((m, \boldsymbol{x}), \mu) \in Steps_{\mathcal{D}}(\mathtt{c})$. These pairs induce corresponding abstract transitions $((m, B), \mu') \in \mathcal{ST}_{\mathcal{D}}(\mathtt{c})$ where B is the abstract state containing $\boldsymbol{x}$, and $\mu' \in lift_{\mathcal{B}}(\mu)$. Then, let $\mathcal{ST}_{\mathcal{D}} = \cup_{\mathtt{c} \in \mathtt{C}} \mathcal{ST}_{\mathcal{D}}(\mathtt{c})$.*

Example 1. Consider Fig. 2 and assume we have a guarded command $\mathtt{c} : condition \rightarrow p_1 : up_1 + \ldots + p_4 : up_4$. Thus $q_{\mathtt{c}} = 4$. The abstract states are represented by circles, labelled with the corresponding tuple. The concrete states are represented by black points, labelled with only the evaluation of the variables (assume that all of them are different). Thus s_0 represents state (m_0, s_0) and so on. Arrows are transitions in the concrete models, where the labels represent the probability p_i of the corresponding update up_i of $\mathtt{c}$.

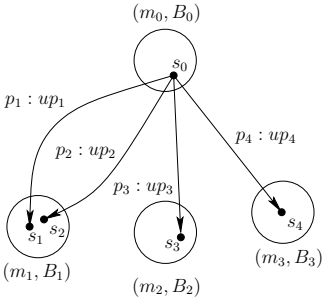


Fig. 2. Illustrating the abstract discrete transitions in the quotient automaton

Consider the two (there may be more) concrete transitions in $T_{ind(\mathcal{H})}$: $((m_0, s_0), (m_1, s_1))$, $((m_0, s_0), (m_1, s_2)) \in T_{\mathcal{D}}$. Both of them lead from (m_0, B_0) to the same abstract state (m_1, B_1). By Definition 7, we have that $((m_0, B_0), (m_i, B_i)) \in \mathcal{T}_{\mathcal{D}}$ for $i = 1, 2, 3$ in $Quo_{ind(\mathcal{H})}(\mathcal{B})$.

We have a concrete transition $((m_0, s_0), \mu)$ where μ is defined by: $\mu(s_i) = p_i$ for $i = 1, 2, 3, 4$. Assume first that B_0, B_1, B_2, B_3 are disjoint. By Definition 8, $lift_{\mathcal{B}}(\mu) = \{\mu'\}$ where μ' is defined by: $\mu'(m_1, B_1) = p_1 + p_2$, $\mu'(m_2, B_2) = p_3$, and $\mu'(m_3, B_3) = p_4$. Then, by Definition 9, this induces an abstract transition $((m_0, B_0), \mu') \in lift_{\mathcal{B}}(\mu)$ in $Quo_{\mathcal{H}}(\mathcal{B})$.

Assume now that the abstract states B_1 and B_2 are not disjoint, and that s_2 is on the common border of (m_1, B_1) and (m_2, B_2) (which implies also $m_1 = m_2$). In this case the set $lift_{\mathcal{B}}(\mu)$ contains another element μ'' which defined by: $\mu''(m_1, B_1) = p_1$, $\mu''(m_2, B_2) = p_2 + p_3$ and $\mu''(m_3, B_3) = p_4$. Again by Definition 9, μ'' induces another abstract transition $((m_0, B_0), \mu'')$ in $Quo_{\mathcal{H}}(\mathcal{B})$.

4.2 Soundness

Given a probabilistic hybrid automaton $\mathcal{H}$ and a set of abstract states $\mathcal{B}$, we defined a probabilistic quotient automaton $Quo_{\mathcal{H}}(\mathcal{B})$. The following lemma shows that this automaton conservatively over-approximates $\mathcal{M}_{\mathcal{H}}$.

Lemma 1. *$Quo_{\mathcal{H}}(\mathcal{B})$ simulates $\mathcal{M}_{\mathcal{H}}$.*

Proof sketch: We define $R = \{((m, \boldsymbol{x}), (m', B)) \in S \times \mathcal{B} \mid m = m' \wedge \boldsymbol{x} \in B\}$. It suffices to show that R is a simulation relation. Let $((m, \boldsymbol{x}), (m, B)) \in R$. The first two conditions for simulation relations are trivially satisfied. It remains the third condition. There are two type of transitions starting from $(m, \boldsymbol{x})$ in $\mathcal{M}_{\mathcal{H}}$: the case $((m, \boldsymbol{x}), Dirac_{(m, \boldsymbol{x}')}) \in Steps_{\mathcal{C}}$ is trivial and skipped. Now consider the case $((m, \boldsymbol{x}), \mu) \in Steps_{\mathcal{D}}$: there exists then a guarded command $\mathtt{c}$ such that $((m, \boldsymbol{x}), \mu) \in Steps_{\mathcal{D}}(\mathtt{c})$. Let $\mathtt{c}$ and $ind(\mathtt{c}) = \{\mathtt{u}_1, \ldots, \mathtt{u}_{q_{\mathtt{c}}}\}$ be as described in Definition 3, and let $(m_i, \boldsymbol{x}_i) = update_i(m, \boldsymbol{x})$ be the state with respect to $update_i$ for $i = 1, \ldots, q_{\mathtt{c}}$. Note that it could be the case that, for $i \neq j$, $\boldsymbol{x_i} = \boldsymbol{x_j}$. Moreover, let $(m_i, B_i) \in \mathcal{B}$ denote the abstract state satisfying $\boldsymbol{x}_i \in B_i$. By construction of the relation R, we know that $((m_i, \boldsymbol{x_i}), (m_i, B_i)) \in R$. By the definition of $\mathcal{ST}$ (cf. Definition 11), we have that $((m, B), \mu') \in \mathcal{ST}_{\mathcal{D}}(\mathtt{c})$ where $\mu'(m_i, B_i) = \sum_{j \in \{j \mid m_j = m_i \wedge B_j = B_i\}} p_j$ for $i = 1, \ldots, q_{\mathtt{c}}$. Define Δ for (μ, μ') with respect to R by: $\Delta((m_i, \boldsymbol{x}_i), (m_i, B_i))$ equals $\mu(m_i, \boldsymbol{x}_i)$ for $i = 1, \ldots, q_{\mathtt{c}}$, and equals 0 otherwise. It remains to show that Δ is the proper weight function. For the first condition, assume $\Delta((m^*, \boldsymbol{x^*}), (m', B')) > 0$. By the definition of Δ, we have $m^* = m'$ and $\boldsymbol{x^*} \in B'$, implying $((m^*, \boldsymbol{x^*}), (m', B')) \in R$. Now we show the third condition (the second condition is similar). Let (m_j, B_j) be an abstract state with $j \in \{1, \ldots, q_{\mathtt{c}}\}$ (otherwise trivial). On one hand, due to the definition of μ', $\mu'(m_j, B_j) = \sum_{i \in I} p_i$ where $I = \{i \mid m_i = m_j \wedge B_i = B_j\}$ denotes the set of all indices i such that $(m_i, B_i) = (m_j, B_j)$. On the other hand, by the definition

of Δ, it holds $\sum_{i\in I} p_i = \sum_{\boldsymbol{x}_k\in B_j} \mu(m_j,\boldsymbol{x}_k) = \sum_{k\in I} \Delta((m_j,\boldsymbol{x}_k),(m_j,B_j))$ (cf. Equation (1)), which implies the third condition. ■

Since simulation on probabilistic automata preserves safety properties [13], we have the correctness of our construction:

Lemma 2. *The abstraction preserves the safety property: if the probability of reaching $UnSafe$ in $Quo_{\mathcal{H}}(\mathcal{B})$ is bounded by ε, this is also the case in $\mathcal{H}$.*

4.3 Abstractions for $\mathcal{H}$

Consider the probabilistic hybrid automaton $\mathcal{H}$. Often the computation of the exact quotient automaton $Quo_{\mathcal{H}}(\mathcal{B})$ as defined in Definition 9 refers to concrete states, and is hard or even impossible. In this subsection we introduce the notion of abstractions which over-approximate the quotient automata. As a convention we use the primed version $\mathcal{T}',\mathcal{I}',\mathcal{U}'$ to denote the set of transitions, initial states, unsafe states in the abstraction.

Definition 10. *Let $\mathcal{H} = (Flow, C, Init, UnSafe)$ be a probabilistic hybrid automaton, and let $\mathcal{B}$ denote the abstract state space. Then,*

- *$Abs_{ind(\mathcal{H})}(\mathcal{B}) = (\mathcal{B},\mathcal{T}',\mathcal{I}',\mathcal{U}')$ is an abstraction of the quotient $Quo_{ind(\mathcal{H})}(\mathcal{B})$ iff $\mathcal{T}' = \cup_{u\in ind(C)}\mathcal{T}'_{\mathcal{D}}(u)\cup\mathcal{T}'_{\mathcal{C}}$ and it holds $\mathcal{T}_{\mathcal{C}} \subseteq \mathcal{T}'_{\mathcal{C}}$, $\mathcal{T}_{\mathcal{D}}(u) \subseteq \mathcal{T}'_{\mathcal{D}}(u)$ for $u \in ind(C)$, $\mathcal{I} \subseteq \mathcal{I}'$ and $\mathcal{U} \subseteq \mathcal{U}'$,*
- *$Abs_{\mathcal{H}}(\mathcal{B}) = (\mathcal{B},\mathcal{ST}',\mathcal{I}',\mathcal{U}')$ is an abstraction of the quotient $Quo_{\mathcal{H}}(\mathcal{B})$ iff $\mathcal{ST}' = \cup_{c\in C}\mathcal{ST}'_{\mathcal{D}}(c)\cup\mathcal{ST}'_{\mathcal{C}}$ and it holds $\mathcal{ST}_{\mathcal{D}}(c) \subseteq \mathcal{ST}'_{\mathcal{D}}(c)$ for $c \in C$, $\mathcal{ST}_{\mathcal{C}} \subseteq \mathcal{ST}'_{\mathcal{C}}$, $\mathcal{I} \subseteq \mathcal{I}'$ and $\mathcal{U} \subseteq \mathcal{U}'$.*

In that case, we say also that $Abs_{ind(\mathcal{H})}(\mathcal{B})$ is an abstraction of the induced hybrid automaton $ind(\mathcal{H})$. Similarly, we say also that $Abs_{\mathcal{H}}(\mathcal{B})$ is an abstraction of the probabilistic hybrid automaton $\mathcal{H}$. Since the abstraction as defined may have more initial states, unsafe states and transitions than the quotient automaton, it is easy to verify that the abstraction simulates the corresponding quotient automaton. Since simulation is transitive, the abstraction also simulates the corresponding semantics automaton. Thus, the abstraction preserves also safety properties of $\mathcal{H}$.

4.4 Computing Abstractions

Let $\mathcal{H}$ be a probabilistic hybrid automaton. Existing methods can be used to compute an abstraction $Abs_{ind(\mathcal{H})}(\mathcal{B})$ for the induced hybrid automaton $ind(\mathcal{H})$, for example [4, 14, 17]. In the following we define an abstraction based on $Abs_{ind(\mathcal{H})}(\mathcal{B})$:

Definition 11. *For a probabilistic hybrid automaton $\mathcal{H}$, let $\mathcal{B}$ be the abstract state space, and $Abs_{ind(\mathcal{H})} = (\mathcal{B},\mathcal{T}'_{\mathcal{D}}\cup\mathcal{T}'_{\mathcal{C}},\mathcal{I}',\mathcal{U}')$ be an abstraction of $ind(\mathcal{H})$. We define $Abs_{\mathcal{H}}(\mathcal{B}) = (\mathcal{B},\mathcal{ST}'_{\mathcal{C}}\cup\mathcal{ST}'_{\mathcal{D}},\mathcal{I}',\mathcal{U}')$ for $\mathcal{H}$ as follows:*

- *$\mathcal{ST}'_{\mathcal{C}} = \mathcal{T}'_{\mathcal{C}}$,*
- *$\mathcal{ST}'_{\mathcal{D}}$ corresponds to the set of abstract transitions due to discrete jumps. We first define the transition induced by one fixed guarded command c : $condition \rightarrow p_1 : update_1 + \ldots + p_{q_c} : update_{q_c}$, and $ind(c) = \{u_1,\ldots,u_{q_c}\}$ as defined in Definition 3. Then, for every sequence of abstract states*

$(m, B), (m_1, B_1), \ldots, (m_{q_c}, B_{q_c})$ satisfying the condition: $((m, B), (m_i, B_i)) \in \mathcal{T}'_{\mathcal{D}}(\mathfrak{u}_i)$ for $i = 1, \ldots, q_c$ we introduce the transition $((m, B), \mu) \in \mathcal{ST}'_{\mathcal{D}}(\mathfrak{c})$ such that $\mu(m_i, B_i) = \sum_{j \in \{j \mid m_j = m_i \wedge B_j = B_i\}} p_j$ for $i = 1, \ldots, q_c$. Then, $\mathcal{ST}'_{\mathcal{D}}$ is defined to be $\bigcup_{\mathfrak{c} \in \mathfrak{C}} \mathcal{ST}'_{\mathcal{D}}(\mathfrak{c})$.

Is $Abs_{\mathcal{H}}(\mathcal{B})$ in fact an abstraction of $\mathcal{H}$? Since $Abs_{ind(\mathcal{H})}(\mathcal{B})$ is an abstraction for $Quo_{ind(\mathcal{H})}(\mathcal{B})$, by Definition 10 it holds that $\mathcal{T}_{\mathcal{C}} \subseteq \mathcal{T}'_{\mathcal{C}}, \mathcal{I} \subseteq \mathcal{I}', \mathcal{U} \subseteq \mathcal{U}'$ and that $\mathcal{T}_{\mathcal{D}}(\mathfrak{u}) \subseteq \mathcal{T}'_{\mathcal{D}}(\mathfrak{u})$ for $\mathfrak{u} \in ind(\mathfrak{C})$. Note that in general most of the inclusions above are strict [4,14]. By the construction of $Abs_{\mathcal{H}}(\mathcal{B})$, it holds that $\mathcal{ST}_{\mathcal{C}} \subseteq \mathcal{ST}'_{\mathcal{C}}, \mathcal{I} \subseteq \mathcal{I}', \mathcal{U} \subseteq \mathcal{U}'$. The following lemma shows that it holds also $\mathcal{ST}_{\mathcal{D}} \subseteq \mathcal{ST}'_{\mathcal{D}}$:

Lemma 3. *Consider the abstraction $Abs_{\mathcal{H}}(\mathcal{B})$ as defined in Definition 11. Then, it holds that $\mathcal{ST}_{\mathcal{D}}(\mathfrak{c}) \subseteq \mathcal{ST}'_{\mathcal{D}}(\mathfrak{c})$, for all $\mathfrak{c} \in \mathfrak{C}$.*

Proof sketch: Fix $\mathfrak{c} \in \mathfrak{C}$. Assume that $((m, B), \mu') \in \mathcal{ST}_{\mathcal{D}}(\mathfrak{c})$. Then, by Definition 9, there exists $\boldsymbol{x} \in B$ and a transition $((m, \boldsymbol{x}), \mu) \in Steps_{\mathcal{D}}(\mathfrak{c})$ such that $\mu' \in lift_{\mathcal{B}}(\mu)$. For $i = 1, \ldots, q_{\mathfrak{c}}$, let $(m_i, \boldsymbol{x_i}) = update_i(m, \boldsymbol{x})$, and let (m_i, B_i) be the abstract states corresponding to the distribution μ' (cf. Definition 8), i.e., $\mu'(m_i, B_i) = \sum_{\{j \mid (m_j, B_j) = (m_i, B_i)\}} \mu(m_j, \boldsymbol{x_j})$. Obviously, $((m, \boldsymbol{x}), (m_i, \boldsymbol{x_i})) \in T_{\mathcal{D}}(\mathfrak{u}_i)$. Since $\boldsymbol{x_i} \in B_i$ it holds that $((m, B), (m_i, B_i)) \in \mathcal{T}_{\mathcal{D}}(\mathfrak{u}_i) \subseteq \mathcal{T}'_{\mathcal{D}}(\mathfrak{u}_i)$ for $i = 1, \ldots, q_{\mathfrak{c}}$. By Definition 11, we have that $((m, B), \mu') \in \mathcal{ST}_{\mathcal{D}}(\mathfrak{c})$. ■

The set of transitions $\mathcal{ST}'_{\mathcal{D}}(\mathfrak{c})$ is indeed an over-approximation, which is illustrated as follows.

Example 2. Consider the fragment of the abstraction depicted in Fig. 3 in which we assume that the transitions correspond to the guarded command $\mathfrak{c}$ with $q_{\mathfrak{c}} = 4$: $condition \rightarrow p_1 : up_1 + \ldots + p_4 : up_4$. The abstract states are represented by circles, labelled with the corresponding tuple. The concrete states are represented by black points, labelled with only the evaluation of the variables (assume that all of them are different). Thus s_0 represents state (m_0, s_0) and so on. Arrows are transitions in the concrete models, where the labels represent the probability p_i of the corresponding update up_i of $\mathfrak{c}$. Assume that all of the concrete states are different and are not on borders. (Note: only parts of successor distributions are depicted, and we assume that other parts (e.g. for state (m_0, s_1)) lead to abstract states outside the depicted fragment.)

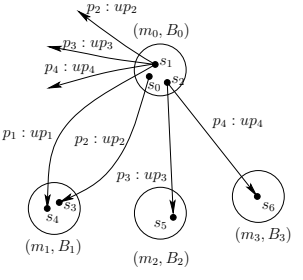


Fig. 3. Abstracting abstract discrete transitions

Now we consider the distribution $\mu^* \in Distr(\mathcal{B})$ which is defined as follows: $\mu^*(m_1, B_1) = p_1 + p_2$, $\mu^*(m_2, B_2) = p_3$ and $\mu^*(m_3, B_3) = p_4$. By the above assumption, no concrete successor distributions of s_0, s_1 or s_2 could induce μ^* according Definition 8. Thus, by Definition 9, $((m_0, B_0), \mu^*) \notin \mathcal{ST}_{\mathcal{D}}(\mathfrak{c})$. On the other hand, it holds $((m_0, B_0), (m_1, B_1)) \in \mathcal{T}'_{\mathcal{D}}(up_i)$ for $i = 1, 2$, $((m_0, B_0), (m_2, B_2)) \in \mathcal{T}'_{\mathcal{D}}(up_3)$, and $((m_0, B_0), (m_3, B_3)) \in \mathcal{T}'_{\mathcal{D}}(up_4)$. Thus, by Definition 11 we have that $((m_0, B_0), \mu^*) \in \mathcal{ST}'_{\mathcal{D}}(\mathfrak{c})$.

Lemma 3 implies that $Abs_{\mathcal{H}}(\mathcal{B})$ is an abstraction of $Quo_{\mathcal{H}}(\mathcal{B})$. Thus:

Theorem 1. *For every probabilistic hybrid automaton H, for every abstraction $Abs_{ind(\mathcal{H})}(\mathcal{B})$ of the induced hybrid automaton $ind(H)$, the safety of $Abs_{\mathcal{H}}(\mathcal{B})$ implies the safety of H.*

5 Experiments

We implemented our method in the prototypical tool `ProHVer` (probabilistic hybrid automata verifier). It combines a modified version of `PHAVer` [17] to obtain the abstract state space with a component to compute an upper probability bound for the reachability problem using value iteration in the induced abstract probabilistic automaton. To show the applicability of our approach, we applied `ProHVer` on several case studies, which are small but diverse in the nature of their behaviour. Even though in each of them we considered bounded reachability (by using a clock variable to bound the time) to get result other than 1, our method is not in principal restricted to time bounded reachability.

`PHAVer` covers the reachable continuous space (per discrete location) by polyhedra of a maximal width. It can split locations (introducing new discrete locations) if the over-approximations carried out while constructing this covering are too coarse. This is effective in practice. But if we attempt to improve precision by reducing the maximal width, the resulting covering and location splits can look entirely different. This carries over to the probabilistic side.

This phenomenon of `PHAVer` may induce situations, where that reduced width setting does not lead to tighter probability bounds. Usually it does.

We here consider the thermostat example depicted in Fig. 4, which is extended from the one in [14]. There are four modes: $Cool$, $Heat$, $Check$ and $Error$. The latter mode models the occurrence of a failure, where the temperature sensor gets stuck at the last checked temperature. The set of variables are $\{t, x, T\}$ where T represents the temperature, t represents a local timer and x is used to measure the total time passed so far. Thus, in all modes it holds that $\dot{x} = 1$ and $\dot{t} = 1$. In each mode there is also an invariant constraint restricting the set of state space for this mode. Invariant constraints are only for the sake of convenience and comparison with [14].

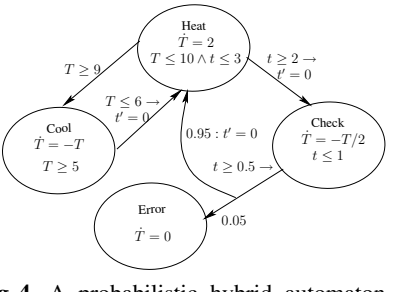


Fig. 4. A probabilistic hybrid automaton for the thermostat

The given initial condition is $m = Heat \wedge t = 0 \wedge x = 0 \wedge 9 \leq T \leq 10$. The unsafe constraint is $m = Error \wedge x \leq 5$, which corresponds to reaching the $Error$ mode within time 5. Assume that the probability threshold for this risk is specified to be 0.2. `ProHVer` can verify this nontrivial system and property, and will answer that the system is safe, the upper bound computed is 0.097.

In Fig. 5, we give probability bounds and performance statistics (time to build the abstraction – the value iteration time is negligible, and number of constructed abstract states) for different time bounds. For the left (right) part we instantiated the splitting

interval for variable x with length 2 (respectively length 10). This governs the refinement technique of PHAVer. The time needed for the analysis as well as the number of states of the abstract transition systems grows about linearly in the time bound, though with oscillations. Comparing the left and the right side, we see that for the larger interval we need less resources, as was to be expected. Due to the way PHAVer splits locations along intervals, for some table entries, we see somewhat counter-intuitive behaviour. We observe that bounds do not necessarily improve with decreasing interval length. This is because PHAVer does not guarantee abstractions with smaller intervals to be an improvement, though they are in most cases. Furthermore, the abstraction we obtain from PHAVer can not guarantee probability bounds to increase monotonically with the time bound. This is because a slightly increased time bound might induce an entirely different abstraction, leading to a tighter probability bound, and thus giving the impression of a decrease in probability, even though the actual maximal probability indeed stays the same or increases.

time	interval length 2			interval length 10		
bound	prob.	build (s)	#states	prob.	build (s)	#states
2	0	0	11	0	0	8
4	0.05	0	43	1	0	12
5	0.097	1	58	1	0	13
20	0.370	20	916	1	1	95
40	0.642	68	2207	0.512	30	609
80	0.884	134	4916	1	96	1717
120	0.940	159	4704	0.878	52	1502
160	0.986	322	10195	0.954	307	4260
180	0.986	398	10760	0.961	226	3768
600	1.0	1938	47609	1	1101	12617

Fig. 5. Thermostat performance

In addition to the thermostat case, we have considered a selection of other case studies: a bouncing ball assembled from different materials, a water level control system where sensor values may be delayed probabilistically, and an autonomous lawn-mower that uses a probability bias to avoid patterns on lawns. As safety problems to be verified we considered (time bounded) reachability properties. We varied the time bounds and other parameters of the analysis, leading to different upper bounds of varying precision. Mostly, the upper bounds we could obtain were tight or exact (checked by manual inspection). Due to space restrictions, we have put the complete descriptions of all case studies and corresponding results on our preliminary homepage for the tool at:

http://depend.cs.uni-sb.de/tools/prohver

6 Conclusions

In this paper we have discussed how to check safety properties for probabilistic hybrid automata. These models and properties are of central importance for the design and verification of emerging wireless and embedded real-time applications. Moreover, being based on arbitrary abstractions computed by tools for the analysis of non-probabilistic hybrid automata, improvements in effectivity of such tools directly carry over to improvements in effectivity of the technique we describe. The applicability of our approach has been demonstrated on a number of case studies, tackled using a prototypical implementation.

As future work we are investigating whether our approach can be adapted to the safety verification problem for more general probabilistic hybrid systems [7,8], that is, systems with stochastic differential equations instead of ordinary differential equations.

Acknowledgement. The work of Lijun Zhang was partially supported in part by MT-LAB, a VKR Centre of Excellence. Part of this work was done while Lijun Zhang was at Saarland University and Computing Laboratory, Oxford University. The work of Zhikun She was partly supported by Beijing Nova Program, the one of Stefan Ratschan has been supported by GAČR grant 201/08/J020 and by the institutional research plan AV0Z100300504. Holger Hermanns and Ernst Moritz Hahn were supported by the NWO-DFG bilateral project ROCKS, by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 AVACS, and by the European Community's Seventh Framework Programme under grant agreement n^o 214755. The work of Ernst Moritz Hahn has been also supported by the Graduiertenkolleg "Leistungsgarantien für Rechnersysteme".

References

1. Alur, R., Courcoubetis, C., Halbwachs, N., Henzinger, T.A., Ho, P.H., Nicollin, X., Olivero, A., Sifakis, J., Yovine, S.: The algorithmic analysis of hybrid systems. Theoretical Computer Science 138, 3–34 (1995)
2. Preußig, J., Kowalewski, S., Wong-Toi, H., Henzinger, T.: An algorithm for the approximative analysis of rectangular automata. In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, p. 228. Springer, Heidelberg (1998)
3. Clarke, E., Fehnker, A., Han, Z., Krogh, B., Stursberg, O., Theobald, M.: Verification of hybrid systems based on counterexample-guided abstraction refinement. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 192–207. Springer, Heidelberg (2003)
4. Ratschan, S., She, Z.: Safety verification of hybrid systems by constraint propagation based abstraction refinement. ACM Transactions on Embedded Computing Systems 6 (2007)
5. Altman, E., Gaitsgory, V.: Asymptotic optimization of a nonlinear hybrid system governed by a markov decision process. SIAM Journal of Control and Optimization 35, 2070–2085 (1997)
6. Sproston, J.: Decidable model checking of probabilistic hybrid automata. In: Joseph, M. (ed.) FTRTFT 2000. LNCS, vol. 1926, pp. 31–45. Springer, Heidelberg (2000)
7. Bujorianu, M.L.: Extended stochastic hybrid systems and their reachability problem. In: Alur, R., Pappas, G.J. (eds.) HSCC 2004. LNCS, vol. 2993, pp. 234–249. Springer, Heidelberg (2004)
8. Bujorianu, M.L., Lygeros, J., Bujorianu, M.C.: Bisimulation for general stochastic hybrid systems. In: Morari, M., Thiele, L. (eds.) HSCC 2005. LNCS, vol. 3414, pp. 198–214. Springer, Heidelberg (2005)
9. Abate, A., Prandini, M., Lygeros, J., Sastry, S.: Probabilistic reachability and safety for controlled discrete time stochastic hybrid systems. Automatica 44, 2724–2734 (2008)
10. Blom, H., Lygeros, J.: Stochastic Hybrid Systems: Theory and Safety Critical Applications. Lecture Notes in Control and Information Sciences, vol. 337. Springer, Heidelberg (2006)
11. Fränzle, M., Hermanns, H., Teige, T.: Stochastic satisfiability modulo theory: A novel technique for the analysis of probabilistic hybrid systems. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 172–186. Springer, Heidelberg (2008)
12. Teige, T., Fränzle, M.: Constraint-based analysis of probabilistic hybrid systems. In: ADHS (2009)
13. Segala, R., Lynch, N.: Probabilistic simulations for probabilistic processes. Nordic Journal of Computing 2, 250–273 (1995)
14. Alur, R., Dang, T., Ivancic, F.: Predicate abstraction for reachability analysis of hybrid systems. ACM Transactions on Embedded Computing Systems 5, 152–199 (2006)

15. D'Argenio, P.R., Jeannet, B., Jensen, H.E., Larsen, K.G.: Reachability analysis of probabilistic systems by successive refinements. In: de Luca, L., Gilmore, S. (eds.) PROBMIV 2001, PAPM-PROBMIV 2001, and PAPM 2001. LNCS, vol. 2165, pp. 39–56. Springer, Heidelberg (2001)
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. 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)
18. Kwiatkowska, M.Z., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science 282, 101–150 (2002)
19. Henzinger, T.A., Kopke, P.W., Puri, A., Varaiya, P.: What's decidable about hybrid automata. Journal of Computer and System Sciences 57, 94–124 (1998)
20. Lafferriere, G., Pappas, G.J., Yovine, S.: A new class of decidable hybrid systems. In: Vaandrager, F.W., van Schuppen, J.H. (eds.) HSCC 1999. LNCS, vol. 1569, pp. 137–151. Springer, Heidelberg (1999)
21. Kwiatkowska, M., Norman, G., Parker, D.: Stochastic games for verification of probabilistic timed automata. In: Ouaknine, J., Vaandrager, F.W. (eds.) FORMATS 2009. LNCS, vol. 5813, pp. 212–227. Springer, Heidelberg (2009)
22. Davis, M.: Markov Models and Optimization. Chapman & Hall, Boca Raton (1993)
23. Arnold, L.: Stochastic Differential Equations: Theory and Applications. Wiley - Interscience, Chichester (1974)
24. Hu, J., Lygeros, J., Sastry, S.: Towars a theory of stochastic hybrid systems. In: Lynch, N.A., Krogh, B.H. (eds.) HSCC 2000. LNCS, vol. 1790, pp. 160–173. Springer, Heidelberg (2000)
25. Bujorianu, M.L., Lygeros, J.: Toward a general theory of stochastic hybrid systems. In: Stochastic Hybrid Systems Theory and Safety Critical Applications, pp. 3–30 (2006)
26. Julius, A.A.: Approximate abstraction of stochastic hybrid automata. In: Hespanha, J.P., Tiwari, A. (eds.) HSCC 2006. LNCS, vol. 3927, pp. 318–332. Springer, Heidelberg (2006)
27. Bujorianu, M.L., Lygeros, J., Langerak, R.: Reachability analysis of stochastic hybrid systems by optimal control. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 610–613. Springer, Heidelberg (2008)
28. Jonsson, B., Larsen, K.G.: Specification and refinement of probabilistic processes. In: LICS, pp. 266–277 (1991)

A Logical Product Approach to Zonotope Intersection

Khalil Ghorbal, Eric Goubault, and Sylvie Putot

Laboratory for the Modelling and Analysis of Interacting Systems
CEA, LIST, Boîte 94, Gif-sur-Yvette, F-91191 France
firstname.lastname@cea.fr

Abstract. We define and study a new abstract domain which is a fine-grained combination of zonotopes with (sub-)polyhedric domains such as the interval, octagon, linear template or polyhedron domains. While abstract transfer functions are still rather inexpensive and accurate even for interpreting non-linear computations, we are able to also interpret tests (i.e. intersections) efficiently. This fixes a known drawback of zonotopic methods, as used for reachability analysis for hybrid systems as well as for invariant generation in abstract interpretation: intersection of zonotopes are not always zonotopes, and there is not even a best zonotopic over-approximation of the intersection. We describe some examples and an implementation of our method in the APRON library, and discuss some further interesting combinations of zonotopes with non-linear or non-convex domains such as quadratic templates and maxplus polyhedra.

1 Introduction

Zonotopic abstractions are known to give fast and accurate over-approximations in invariant synthesis for static analysis of programs, as introduced by the authors [10, 11, 7], as well as in reachability analysis of hybrid systems [8]. The main reason for this is that the interpretation of linear assignments is exact and done in linear time in terms of the "complexity" of the zonotopes, and non-linear expressions are dynamically linearized in a rather inexpensive way, unlike for most of other sub-polyhedric domains (zones [19], linear templates [21], even polyhedra [6]). But unions, at the exception of recent work [14], and more particularly intersections [9] are not canonical operations, and are generally computed using approximate and costly methods, contrarily to the other domains we mentioned. We present in this article a way to combine the best of the two worlds: by constructing a form of logical product [15] of zonotopes with any of these sub-polyhedric domains, we still get accurate and inexpensive methods to deal with the interpretation of linear and non-linear assignments, while intersections in particular, come clean thanks to the sub-polyhedric component of the domain.

Consider for instance the following program (loosely based on non-linear interpolation methods in e.g. embedded systems), which will be our running example:

```
real x = [0,10];
real y = x*x - x;
if (y >= 0) y = x/10; /* (x=0 or x >= 1) and y in [0,1] */
else y = x*x+2;       /* (x>0 and x<1) and y in [2,3] */
```

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 212–226, 2010.

As indicated in the comments of the program, the `if` branch is taken when we have $x = 0$ or $x \geq 1$, so that y at the end of the program, is always in $[0, 3]$. Although this program looks quite simple, it is difficult to analyze, and the invariants found for y at the end of the program by classical domains[1] are disappointing: intervals, octagons, polyhedra, or zonotopes without constraint all find a range of values for y larger or equal than $[0, 102]$: even those which interpret quite accurately non-linear operations are not able to derive a constraint on x from the constraint on y. Whereas by the method proposed here, a logical product of zonotopes with intervals, in its APRON implementation, we find the much better range $[0, 9.72]$ (comparable to the exact result $[0, 3]$).

Contents of the paper. We first introduce in Section 2 affine sets, a zonotopic abstract domain for abstract interpretation, that abstracts input/output relations in a program. We then introduce the problem of computing intersections in Section 3: starting with the running example, we define constrained affine sets as the combination of zonotopes with polyhedric domains and show they are well suited for the interpretation of tests. We then generalize the order on affine sets to constrained affine sets and define monotonic abstract transfer functions for arithmetic operators, that over-approximate the concrete semantics. Section 4 completes the definition of this new abstract domain: starting with the easier "one-variable" problem, we then give an algorithm for computing a join operator. We demonstrate the interest of the domain by describing in Section 5 the results on some examples, based on an implementation of our method in the library APRON. We conclude by a discussion of future work, including some further interesting combinations of zonotopes with non-linear or non-convex domains such as quadratic templates and maxplus polyhedra.

Related work. In [17], the authors propose an approach based on a reduced product [5], to get more tractable and efficient methods for deriving sub-polyhedric invariants. But, still, the reduction algorithm of [17] is fairly expensive, and this domain also suffers from the drawbacks of polyhedra, in the sense that it is not well suited for efficiently and precisely deriving invariants for non-linear computations. Logical products in abstract interpretation are defined in [15]. The authors use the Nelson-Oppen combination method for logical theories, in the convex case, to get polynomial time abstractions on a much finer (than classical reduced products) combination of two abstract domains. As explained in Section 3.2, this approach does not directly carry over our case, because the theories we want to combine do not satisfy all the hypotheses of [15]. We thus choose in this paper a direct approach to the logical product of zonotopes with other classical abstract domains.

2 Affine Sets: Main Definitions and Properties

2.1 Affine Arithmetic and Zonotopes

Affine arithmetic is an extension of interval arithmetic on affine forms, first introduced in [4], that takes into account affine correlations between variables. An *affine form* is a

[1] The experiments were carried out using the domains interfaced within APRON [20].

formal sum over a set of *noise symbols* ε_i

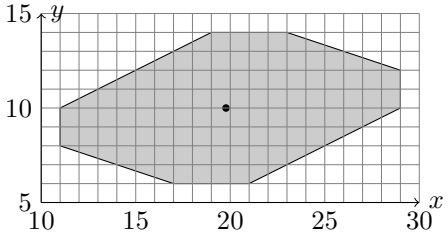
$$\hat{x} \stackrel{\text{def}}{=} \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i,$$

with $\alpha_i^x \in \mathbb{R}$ for all i. Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity $\hat{x}$, its value is unknown but bounded in [-1,1]; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The same noise symbol can be shared by several quantities, indicating correlations among them. These noise symbols can not only model uncertainty in data or parameters, but also uncertainty coming from computation. The semantics of affine operations is straightforward, non affine operations are linearized and introduce a new noise symbol: we refer the reader to [11, 13] for more details.

In what follows, we introduce matrix notations to handle tuples of affine forms. We note $\mathcal{M}(n,p)$ the space of matrices with n lines and p columns of real coefficients. A tuple of affine forms expressing the set of values taken by p variables over n noise symbols $\varepsilon_i,\ 1 \leq i \leq n$, can be represented by a matrix $A \in \mathcal{M}(n+1,p)$. We formally define the zonotopic concretization of such tuples by :

Definition 1. *Let a tuple of affine forms with p variables over n noise symbols, defined by a matrix $A \in \mathcal{M}(n+1,p)$. Its concretization is the zonotope*

$$\gamma(A) = \left\{ {}^tA\,{}^te \mid e \in \mathbb{R}^{n+1}, e_0 = 1, \|e\|_\infty = 1 \right\} \subseteq \mathbb{R}^p \ .$$



For example, for $n = 4$ and $p = 2$, the gray zonotope is the concretisation of the affine set $X = (\hat{x}, \hat{y})$, with $\hat{x} = 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4$, $\hat{y} = 10 - 2\varepsilon_1 + \varepsilon_2 - \varepsilon_4$, and ${}^tA = \begin{pmatrix} 20 & -4 & 0 & 2 & 3 \\ 10 & -2 & 1 & 0 & -1 \end{pmatrix}$.

2.2 An Ordered Structure: Affine Sets

In order to construct an ordered structure preserving abstract input/output relations [14], we now define affine sets X as Minkowski sums of a *central* zonotope, $\gamma(C^X)$ and of a *perturbation* zonotope centered on 0, $\gamma(P^X)$. Central zonotopes depend on central noise symbols ε_i, whose interpretation is fixed once and for all in the whole program: they represent the uncertainty on input values to the program, with which we want to keep as many relations as possible. Perturbation zonotopes depend on perturbation symbols η_j which are created along the interpretation of the program and represent the uncertainty of values due to the control-flow abstraction, for instance while computing the join of two abstract values.

Definition 2. *We define an affine set X by the pair of matrices $(C^X, P^X) \in \mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. The affine form $\pi_k(X) = c_{0k}^X + \sum_{i=1}^n c_{ik}^X \varepsilon_i + \sum_{j=1}^m p_{jk}^X \eta_j$, where the ε_i are the central noise symbols and the η_j the perturbation or union noise symbols, describes the kth variable of X.*

We define an order on affine sets [7, 14] which is slightly more strict than concretization inclusion: it formalizes the fact that the central symbols have a specific interpretation as parametrizing the initial values of input arguments to the analyzed program:

Definition 3. *Let* $X = (C^X, P^X)$, $Y = (C^Y, P^Y)$ *be two affine sets in* $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. *We say that* $X \leq Y$ *iff*

$$\forall u \in \mathbb{R}^p,\ \|(C^Y - C^X)u\|_1 \leq \|P^Y u\|_1 - \|P^X u\|_1 \ .$$

It expresses that the norm of the difference $(C^Y - C^X)u$ for all $u \in \mathbb{R}^p$ is less than what the perturbation terms P^X and P^Y allow, that is the difference of the norms of $P^Y u$ with $P^X u$.

Classically, input/output functional abstractions are handled by adding slack variables corresponding to the initial values of the inputs. Here, we want relations between the variables of the program and the uncertain inputs, that is the inputs that create noise symbols. It can be proved that the relation of Definition 3 is equivalent to the geometric order on the larger zonotopes obtained by adding these slack variables to the zonotopes represented by our affine sets.

The binary relation $\leq$ of Definition 3 is a preorder, that we identify in the sequel with the partial order, quotient of this preorder by the equivalence relation[2] $X \sim Y$ iff by definition $X \leq Y$ and $Y \leq X$. Note also that this partial order is decidable, with a complexity bounded by a polynomial in p and an exponential in $n + m$. In practice, see [14], we do not need to use this costly general decision procedure.

3 *Constrained* Affine Sets for Intersection

We now introduce the logical product of the domain $\mathcal{A}_1$ of Section 2 with any lattice, $(\mathcal{A}_2, \leq_2, \cup_2, \cap_2)$, used to abstract the values of the noise symbols ε_i and η_j. Formally, supposing that we have $n + 1$ noise symbols ε_i and m noise symbols η_j as in Section 2.2, we are given a concretization function: $\gamma_2 : \mathcal{A}_2 \to \mathcal{P}(\{1\} \times \mathbb{R}^n \times \mathbb{R}^m)$ and pseudo-inverse α_2. We now define constrained affine sets:

Definition 4. *A constrained affine set* U *is a pair* $U = (X, \Phi^X)$ *where* $X = (C^X, P^X)$ *is an affine set, and* Φ^X *is an element of* $\mathcal{A}_2$. *Equivalently, we write* $U = (C^X, P^X, \Phi^X)$.

Classical abstractions of "constraints" on the ε_i we will be using throughout this text are $\mathcal{A}$ consisting of products of $1 + n + m$ intervals (with the first one always being equal to 1), zones, octagons, and polyhedra (in the hyperplane $\varepsilon_0 = 1$).

3.1 Interpretation of Tests

Equality tests on variables. We first consider the case of the interpretation of equality test of two variables within an abstract state. Let us begin by a motivating example, which will make clear what the general interpretation of Definition 5 should be.

[2] It can be characterized by $C^X = C^Y$ and same concretizations for P^X and P^Y.

Example 1. Consider, with an interval domain for the noise symbols, $Z = [\![x_1 == x_2]\!]X$ where

$$\begin{cases} \Phi^X = 1 \times [-1,1] \times [-1,1] \times [-1,1] \\ \hat{x}_1^X = 4 + \varepsilon_1 + \varepsilon_2 + \eta_1, \quad \gamma(\hat{x}_1) = [1,7] \\ \hat{x}_2^X = -\varepsilon_1 + 3\varepsilon_2, \quad \gamma(\hat{x}_2) = [-4,4] \end{cases}$$

We look for $\hat{z} = \hat{x}_1 = \hat{x}_2$, with $\hat{z} = z_0 + z_1\varepsilon_1 + z_2\varepsilon_2 + z_3\eta_1$. Using $\hat{x}_1 - \hat{x}_2 = 0$, i.e.

$$4 + 2\varepsilon_1 - 2\varepsilon_2 + \eta_1 = 0, \tag{1}$$

and substituting η_1 in $\hat{z} - \hat{x}_1 = 0$, we deduce $z_0 = 4z_3$, $z_1 = 2z_3 - 1$, $z_2 = -2z_3 + 3$. The abstraction in intervals of constraint (1) yields tighter bounds on the noise symbols: $\Phi^Z = 1 \times [-1,-0.5] \times [0.5,1] \times [-1,0]$. We now look for z_3 that minimizes the width of the concretization of z, that is $0.5|2z_3 - 1| + 0.5|3 - 2z_3| + |z_3|$. A straightforward $O((m+n)^2)$ method to solve the problem evaluates this expression for z_3 successively equal to 0, 0.5 and 1.5: the minimum is reached for $z_3 = 0.5$. We then have

$$\begin{cases} \Phi^Z = 1 \times [-1,-0.5] \times [0.5,1] \times [-1,0] \\ \hat{x}_1^Z = \hat{x}_2^Z (= \hat{z}) = 2 + 2\varepsilon_2 + 0.5\eta_1, \quad \gamma(\hat{x}_1^Z) = \gamma(\hat{x}_2^Z) = [2.5,4] \end{cases}$$

Note that the concretization $\gamma(\hat{x}_1^Z) = \gamma(\hat{x}_2^Z)$ is not only better than the intersection of the concretizations $\gamma(\hat{x}_1^X)$ and $\gamma(\hat{x}_2^X)$ which is $[1,4]$, but also better than the intersection of the concretization of affine forms $(\hat{x}_1^X)$ and $(\hat{x}_2^X)$ for noise symbols in Φ^Z. Note that there is not always a unique solution minimizing the width of the concretization.

In the following, we use bold letters to denote intervals, and for an interval $\boldsymbol{u} = [\underline{u}, \overline{u}]$, we note $\mathrm{dev}(\boldsymbol{u}) = \overline{u} - \underline{u}$.

Definition 5. *Let $X = (C^X, P^X, \Phi^X)$ a constrained affine set with $(C^X, P^X) \in \mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. We define $Z = [\![x_j == x_i]\!]X$ by:*
- $\Phi^Z = \Phi^X \bigcap \alpha_2 \left(\left\{ (\varepsilon_1, \ldots, \varepsilon_n, \eta_1, \ldots, \eta_m) \mid (c_{0j}^X - c_{0i}^X) + \sum_{r=1}^n (c_{rj}^X - c_{ri}^X)\varepsilon_r + \sum_{r=1}^m (p_{rj}^X - p_{ri}^X)\eta_r = 0 \right\} \right)$,
- $c_{rl}^Z = c_{rl}^X, \forall r \in \{0, \ldots, n\}$, and $\forall l \in \{1, \ldots, p\}, l \neq i, j$,
- $p_{rl}^Z = p_{rl}^X, \forall r \in \{1, \ldots, m\}$ and $\forall l \in \{1, \ldots, p\}, l \neq i, j$.
Let k such that $c_{kj}^X - c_{ki}^X \neq 0$, we define

$$c_{li}^Z = c_{lj}^Z = c_{li}^X + \frac{c_{lj}^X - c_{li}^X}{c_{kj}^X - c_{ki}^X}(c_{ki}^Z - c_{kj}^X) \; \forall l \in \{0, \ldots, n\}, \; l \neq k, \tag{2}$$

$$p_{li}^Z = p_{lj}^Z = p_{li}^X + \frac{p_{lj}^X - p_{li}^X}{c_{kj}^X - c_{ki}^X}(c_{ki}^Z - c_{kj}^X) \; \forall l \in \{1, \ldots, m\}, \tag{3}$$

with c_{ki}^Z that minimizes $\sum_{l=1}^n |c_{li}^Z| \, \mathrm{dev}(\boldsymbol{\varepsilon_l^Z}) + \sum_{l=1}^m |p_{li}^Z| \, \mathrm{dev}(\boldsymbol{\eta_l^Z})$.
If for all k, $c_{kj}^X = c_{ki}^X$, then we look for r such that $p_{rj}^X - p_{ri}^X \neq 0$; if for all r, $p_{rj}^X = p_{ri}^X$ then $x_i = x_j$ and $Z = X$.

This expresses that the abstraction of the constraint on the noise symbols induced by the test is added to the domain of constraints, and the exact constraint is used to define

an affine form z satisfying $z = x_j^Z = x_i^Z$, and such that $\gamma(z)$ is minimal. Indeed, let k such that $c_{kj}^X - c_{ki}^X \neq 0$, then $x_j == x_i$ allows to express ε_k as

$$\varepsilon_k = c_{0j}^X - c_{0i}^X + \sum_{1\leq l\leq n, l\neq k} \frac{(c_{lj}^X - c_{li}^X)}{c_{ki}^X - c_{kj}^X}\varepsilon_l + \sum_{1\leq l\leq m} \frac{(p_{lj}^X - p_{li}^X)}{c_{ki}^X - c_{kj}^X}\eta_l \ . \tag{4}$$

We now look for $\pi_i(Z) = \pi_j(Z)$ equal to $\pi_i(X)$ and $\pi_j(X)$ under condition (4) on the noise symbols (where $\pi_k(X)$ describes the kth variable of X, as introduced in Definition 2). Substituting ε_k in for example $\pi_i(Z) = \pi_i(X)$, we can express, for all l, c_{li}^Z and p_{li}^Z as functions of c_{ki}^Z and get possibly an infinite number of solutions defined by (2) and (3) that are all equivalent when (4) holds. When condition (4) will be abstracted in a noise symbols abstract domain such as intervals, these abstract solutions will no longer be equivalent, we choose the one that minimizes the width of $\gamma(\pi_i(Z))$ which is given by $\sum_{l=1}^n |c_{li}^Z| \operatorname{dev}(\boldsymbol{\varepsilon_l^Z}) + \sum_{l=1}^m |p_{li}^Z| \operatorname{dev}(\boldsymbol{\eta_l^Z})$. This sum is of the form $\sum_{l=1}^{m+n} |a_l + b_l c_{ki}^Z|$, with known constants a_l and b_l. The minimization problem can be efficiently solved in $O((m+n)log(m+n))$ time, $m+n$ being the number of noise symbols appearing in the expressions of x_i and x_j, by noting that the minimum is reached for $c_{ki}^Z = -\frac{a_{l_0}}{b_{l_0}}$ for a $l_0 \in \{1, \ldots, m+n\}$. When it is reached for two indexes l_p and l_q, it is reached for all c_{ki}^Z in $\left[-\frac{a_{l_p}}{b_{l_p}}, -\frac{a_{l_p}}{b_{l_p}}\right]$, but we choose one of the bounds of this intervals because it corresponds to the substitution in x_i^Z of one of the noise symbols, and is in the interest for the interpretation of tests on expressions.

Equality tests on expressions. Now, in the case of an equality test between arithmetic expressions, new constraints on the noise symbols can be added, corresponding to the equality of the two expressions interpreted as affine forms. We also choose new affine forms for variables appearing in the equality test: let $X = (C^X, P^X, \Phi^X)$ a constrained affine set with $(C^X, P^X) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. We define $Z = [\![exp1 == exp2]\!]X$ by: $Y_1 = [\![x_{p+1} = exp1]\!][\![x_{p+2} = exp2]\!]X$ using the semantics for arithmetic operations, as defined in section 3.3, then $Y_2 = [\![x_{p+1} == x_{p+2}]\!]Y_1$. Noting that one of the noise symbols appearing in the constraint introduced by the equality test, does not appear in $x_{p+1}^{Y_2} = x_{p+2}^{Y_2}$ as computed by Definition 5, using this constraint we substitute this noise symbol in the other variables in Y_2. We then eliminate the added variables x_{p+1} and x_{p+2} to obtain Z, in which $exp1 == exp2$ is thus algebraically satisfied.

Example 2. Consider $Z = [\![x_1 + x_2 == x_3]\!]X$ where

$$\begin{cases} \Phi^X = 1 \times [-1,1] \times [-1,1] \times [-1,1] \\ \hat{x}_1^X = 2 + \varepsilon_1, \quad \gamma(\hat{x}_1) = [1,3] \\ \hat{x}_2^X = 2 + \varepsilon_2 + \eta_1, \quad \gamma(\hat{x}_2) = [0,4] \\ \hat{x}_3^X = -\varepsilon_1 + 3\varepsilon_2, \quad \gamma(\hat{x}_3) = [-4,4] \end{cases}$$

We first compute $x_4 := x_1 + x_2$ in affine arithmetic: here, problem $x_4 == x_3$ is then the test we solved in example 1. The abstraction in intervals of constraint (1) yields $\Phi^Z = 1 \times [-1, -0.5] \times [0.5, 1] \times [-1, 0]$, and an affine form x_3^Z optimal in the sense of the width of its concretization, $x_3^Z = 2 + 2\varepsilon_2 + 0.5\eta_1$. Now, $\hat{x}_1^X + \hat{x}_2^X = \hat{x}_3^Z$ is satisfied when constraint (1) holds exactly, but not in its interval abstraction Φ^Z. But substituting

ε_1 which does not appear in x_3^Z by $-2+\varepsilon_2-0.5\eta_1$ in $\hat{x}_1^X$ and $\hat{x}_2^X$, we obtain forms $\hat{x}_1^Z$ and $\hat{x}_2^Z$ that satisfy $x_1+x_2 == x_3$ in the abstract domain:

$$\begin{cases} \Phi^Z = 1\times[-1,-0.5]\times[0.5,1]\times[-1,0] \\ \hat{x}_1^Z = \varepsilon_2 - 0.5\eta_1, \quad \gamma(\hat{x}_1) = [0.5,1.5] \\ \hat{x}_2^Z = 2+\varepsilon_2+\eta_1, \quad \gamma(\hat{x}_2) = [1.5,3] \\ \hat{x}_3^Z = 2+2\varepsilon_2+0.5\eta_1, \quad \gamma(\hat{x}_1^Z) = \gamma(\hat{x}_2^Z) = [2.5,4] \end{cases}$$

Inequality tests. In the case of inequality tests, we only add constraints on noise symbols, for example for strict inequality:

Definition 6. *Let* $X = (C^X, P^X, \Phi^X)$ *a constrained affine set with* $(C^X, P^X) \in \mathcal{M}(n+1,p)\times\mathcal{M}(m,p)$. *We define* $Z = [\![exp1 < exp2]\!]X$ *by* $Z = (C^X, P^X, \Phi^Z)$*:*
$\Phi^Z = \Phi^X \bigcap \alpha_2\left(\left\{(\varepsilon_1,\ldots,\varepsilon_n,\eta_1,\ldots,\eta_m) \mid (c^Y_{0p+2}-c^Y_{0p+1}) + \sum_{k=1}^n (c^Y_{kp+2}-c^Y_{kp+1})\varepsilon_k + \sum_{k=1}^m (p^Y_{kp+2}-p^Y_{kp+1})\eta_k < 0\right\}\right)$,
where $Y = [\![x_{p+1} = exp1]\!][\![x_{p+2} = exp2]\!]X$.

3.2 Order Relation

In a standard reduced product [5] of $\mathcal{A}_1$ with $\mathcal{A}_2$, the order relation would naturally be based on the component-wise ordering. But in such products, we cannot possibly reduce the abstract values so that to gain as much collaboration as needed between $\mathcal{A}_1$ and $\mathcal{A}_2$ for giving formal grounds to the reasoning of Example 1 for instance. What we really need is to combine the *logical theories* of affine sets, $Th(\mathcal{A}_1)$[3] with the one of quantifier-free linear arithmetic [18] over the reals, , $Th(\mathcal{A}_2)$[4], including all the domains we have in mind in this paper (intervals, zones, octagons, linear and non-linear templates, polyhedra). Look back at Example 1: we found a solution to the constraint $x_1 == x_2$ via a fine-grained interaction between the two theories $Th(\mathcal{A}_1)$ and $Th(\mathcal{A}_2)$. Unfortunately, the methods of [15] are not directly applicable; in particular $\mathcal{A}_1$ is not naturally expressible as a logical lattice - it is not even a lattice in general. Also, the signatures $\Sigma_{\mathcal{A}_1}$ and $\Sigma_{\mathcal{A}_2}$ share common symbols, which is not allowed in the approach of [15].

In order to compute the abstract transfer functions in the logical product $Th(\mathcal{A}_1)\cup Th(\mathcal{A}_2)$, we first define an order relation on the product domain $\mathcal{A}_1\times\mathcal{A}_2$, that allows a fine interaction between the two domains. First, $X = (C^X, P^X, \Phi^X) \leq Y = (C^Y, P^Y, \Phi^Y)$ should imply that $\Phi^X \leq_2 \Phi^Y$, i.e. the range of values that noise symbols can take in form X is smaller than for Y. Then, we mean to adapt Definition 3 for noise symbols no longer defined in $[-1,1]$ as in the unconstrained case, but in the range of values Φ^X common to X and Y. Noting that:

$$\|C^X u\|_1 = \sup_{\epsilon_i\in[-1,1]} |\langle \epsilon, C^X u\rangle|,$$

where $\langle .,. \rangle$ is the standard scalar product of vectors in $\mathbb{R}^{n+1}$, we set:

[3] Signature $\Sigma_{\mathcal{A}_1}$ comprises equality, addition, multiplication by real numbers and real numbers.
[4] Signature $\Sigma_{\mathcal{A}_2}$ comprises $\Sigma_{\mathcal{A}_1}$ plus inequality and negation.

Definition 7. *Let X and Y be two constrained affine sets. We say that $X \leq Y$ iff $\Phi^X \leq_2 \Phi^Y$ and, for all $t \in \mathbb{R}^p$,*

$$\sup_{(\epsilon,-)\in\gamma_2(\Phi^X)} |\langle (C^Y - C^X)t, \epsilon\rangle| \leq \sup_{(-,\eta)\in\gamma_2(\Phi^Y)} |\langle P^Y t, \eta\rangle| - \sup_{(-,\eta)\in\gamma_2(\Phi^X)} |\langle P^X t, \eta\rangle| \ .$$

The binary relation defined in Definition 7 is a preorder on constrained affine sets which coincides with Definition 3 in the "unconstrained" case when $\Phi^X = \Phi^Y = \{1\} \times [-1,1]^{n+m}$. We use in the sequel its quotient by its equivalence relation, i.e. the partial order generated by it.

Definition 8. *Let X be a constrained affine set. Its concretization in $\mathcal{P}(\mathbb{R}^p)$ is*

$$\gamma(X) = \left\{ {}^tC^X\epsilon + {}^tP^X\eta \mid \epsilon, \eta \in \gamma_2(\Phi^X) \right\} \ .$$

For Φ^X such that $\gamma_2(\Phi^X) = \{1\} \times [-1,1]^{n+m}$, this is equivalent to the concretization of the affine set (C^X, P^X) as defined in Section 2.2. As for affine sets [14], the order relation of Definition 7 is stronger than the geometric order: if $X \leq Y$ then $\gamma(X) \subseteq \gamma(Y)$. This allows for expressing functional dependencies between the input and current values of each variables as discussed in [14].

Note that γ is in general computable when $\mathcal{A}$ is a subpolyhedric domain (intervals, zones, octagons, linear templates and general polyhedra), as a linear transformation applied to a polyhedron. In the same case, the interval concretisation of X can be computed using any (guaranteed) solver for linear programs such as LURUPA [16], since it involves $2p$ (for p variables) linear programs:

$$\sup_{\epsilon,\eta\in\gamma(\Phi^X)} {}^tC^X\epsilon + {}^tP^X\eta, \text{ and } \inf_{\epsilon,\eta\in\gamma(\Phi^X)} {}^tC^X\epsilon + {}^tP^X\eta \ .$$

Of course, when $\mathcal{A}$ is the domain of intervals, this is done by a direct and easy calculation.

3.3 Semantics of Arithmetic Operations

Operations are not different than the ones generally defined on zonotopes, or on affine forms, see [4, 14], the only difference is in the multiplication where we use the constraints on ε_i and η_j to derive bounds for the non-linear part.

We note $[\![\text{new } \varepsilon_{n+1}]\!]_{\mathcal{A}_2}\Phi^X$ the creation of a new noise symbol ε_{n+1} with (concrete) values in $[-1,1]$. We first define the assignment of a new variable x_{p+1} with a range of value $[a,b]$:

Definition 9. *Let $X = (C^X, P^X, \Phi^X)$ be a constrained affine set with $(C^X, P^X) \in \mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$ and $a, b \in \mathbb{R}$. We define $Z = [\![x_{p+1} = [a,b]]\!]X$ where $(C^Z, P^Z) \in \mathcal{M}(n+2, p+1) \times \mathcal{M}(m, p+1)$ with : $\Phi^Z = [\![\text{new } \varepsilon_{n+1}]\!]_{\mathcal{A}_2}\Phi^X$, $C^Z =$*

$$\left(\begin{array}{c|c} C^X & \begin{array}{c}\frac{a+b}{2}\\ 0\\ \dots\\ 0\end{array} \\ \hline 0 & \frac{|a-b|}{2} \end{array}\right), \ P^Z = \left(\begin{array}{c|c} P^X & \begin{array}{c}0\\ \dots\\ 0\end{array} \end{array}\right) \ .$$

We carry on by addition, or more precisely, the operation interpreting the assignment $x_{p+1} := x_i + x_j$ and adding new variable x_{p+1} to the affine set:

Definition 10. *Let $X = (C^X, P^X, \Phi^X)$ be a constrained affine set where (C^X, P^X) is in $\mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. We define $Z = [\![x_{p+1} = x_i + x_j]\!]X = (C^Z, P^Z, \Phi^Z)$ where $(C^Z, P^Z) \in \mathcal{M}(n+1, p+1) \times \mathcal{M}(m, p+1)$ by $\Phi^Z = \Phi^X$ and*

$$C^Z = \left(C^X \middle| \begin{matrix} c^X_{0,i} + c^X_{0,j} \\ \dots \\ c^X_{n,i} + c^X_{n,j} \end{matrix} \right) \text{ and } P^Z = \left(P^X \middle| \begin{matrix} p^X_{1,i} + p^X_{1,j} \\ \dots \\ p^X_{m,i} + p^X_{m,j} \end{matrix} \right) .$$

The following operation defines the multiplication of variables x_i and x_j, appending the result to the constrained affine set X. All polynomial assignments can be defined using this and the previous operations.

Definition 11. *Let $X = (C^X, P^X, \Phi^X)$ be a constrained affine set where (C^X, P^X) is in $\mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. We define $Z = (C^Z, P^Z, \Phi^Z) = [\![x_{p+1} = x_i \times x_j]\!]X$ where $(C^Z, P^Z) \in \mathcal{M}(n+2, p+1) \times \mathcal{M}(m+1, p+1)$ by :*

- $\Phi^Z = [\![\text{new } \varepsilon_{n+1}]\!]_{\mathcal{A}_2} \circ [\![\text{new } \eta_{m+1}]\!]_{\mathcal{A}_2} \Phi^X$
- $c^z_{l,k} = c^x_{l,k}$ *and* $c^z_{n+1,k} = 0$ *for all* $l = 0, \dots, n$ *and* $k = 1, \dots, p$
- *Let m_r (resp. μ_r) be the $(r+1)$th coordinate (i.e. corresponding to ε_r) of* $\text{mid}(\gamma(\Phi^X))$ *(resp. of* $\text{dev}(\gamma(\Phi^X))$*), where* mid *(resp.* dev*) denotes the middle (resp. the radius) of an interval, q_l (resp. χ_l) be the $(l+n+1)$th coordinate (i.e. corresponding to η_l) of* $\text{mid}(\gamma(\Phi^X))$ *(resp. of* $\text{dev}(\gamma(\Phi^X))$*). Write* $d^x_i = c^x_{0,i} + \sum_{1 \le r \le n} c^x_{r,i} m_r + \sum_{1 \le l \le m} p^x_{l,i} q_l$*:*
  $c^z_{0,p+1} = d^x_i d^x_j - \sum_{1 \le r \le n} (d^x_i c^x_{r,j} + d^x_j c^x_{r,i}) m_r - \sum_{1 \le l \le m} (d^x_i p^x_{l,j} + d^x_i p^x_{l,i}) q_l + \sum_{1 \le r \le n} \frac{1}{2} c^x_{r,i} c^x_{r,j} \mu^2_r + \sum_{1 \le l \le m} \frac{1}{2} p^x_{l,i} p^x_{l,j} \chi^2_l$
- $c^z_{l,p+1} = d^x_i c^x_{l,j} + c^x_{l,i} d^x_j$ *for all* $l = 1, \dots, n$
- $c^z_{n+1,p+1} = \sum_{1 \le r \le n} \frac{1}{2} |c^x_{r,i} c^x_{r,j}| \mu^2_r + \sum_{1 \le r \ne l \le n} |c^x_{r,i} c^y_{l,j}| \mu_r \mu_l$
- $p^z_{l,k} = p^x_{l,k}$, $p^z_{m+1,k} = 0$ *and* $p^z_{l,p+1} = 0$*, for all* $l = 1, \dots, m$ *and* $k = 1, \dots, p$
- $p^z_{m+1,p+1} = \sum_{1 \le l \le m} |p^x_{l,i} p^x_{l,j}| \chi^2_l + \sum_{1 \le r \ne l \le m} |p^x_{r,i} p^x_{l,j}| \chi_r \chi_l + \sum_{0 \le r \le n}^{1 \le l \le m} (|c^x_{r,i} p^x_{l,j}| + |p^x_{l,i} c^x_{r,j}|) \mu_r \chi_l$.

The correctness of this abstract semantics stems from the fact that these operations are increasing functions over the set of constrained affine sets. For sub-polyhedric domains $\mathcal{A}_2$, m_r, q_l, μ_r and χ_l are easily computable, solving with a guaranteed linear solver the four linear programming problems $\sup_{\epsilon, \eta \in \gamma(\Phi^X)} \varepsilon_r$ (resp. inf) and $\sup_{\epsilon, \eta \in \gamma(\Phi^X)} \eta_l$ (resp. inf) - for an interval domain for $\mathcal{A}_2$, no such computation is needed of course.

Getting back to the running example of Section 1, in the `false` branch of the `if (y>=0)` test, we have to compute $y = x * x + 2$ with $x = 5 + 5\varepsilon_1$ and $\varepsilon_1 \in [-1, -0.444]$. Using Definition 11 which takes advantage of the bounds on ε_1 to get a better bound on the non-linear part (typically not possible if we had constructed a reduced product), we get $y = 14.93 + 13.9\varepsilon_1 + 0.96\varepsilon_3$ with $\varepsilon_3 \in [-1, 1]$. This gives $\gamma(y) = [0.07, 9.72]$, which is very precise since $\gamma(x) = [0, 2.77]$, hence we should ideally find $\gamma(y)$ in $\gamma(x) * \gamma(x) + 2 = [2, 9.72]$. Note that the multiplication given in Definition 11 and used here, is not the direct adaptation of the multiplication in the unconstrained case, that would give the much less accurate form $y = 41.97 + 50\varepsilon_1 + 10.03\varepsilon_3$:

the better formulation is obtained by choosing an affine form that is a linearization of $x_i \times x_j$ no longer at 0, but at the center of the range of the constrained noise symbols.

4 Join Operator on Constrained Affine Sets

We first examine the easier case of finding a join operator for affine sets with just one variable, and $\mathcal{A}_2$ being the lattice of intervals. We then use the characterisations we find in this case to give efficient formulas for a precise (although over-approximated) join operator in the general case. We do not study here maximal lower bounds of affine sets, although they are naturally linked to the interpretation of tests, Section 3.1, this is outside the scope of this paper.

4.1 The One-Dimensional Case

In dimension one, constrained affine sets are simply *constrained affine forms*:

$$\hat{a} = \left(\hat{a}(\epsilon) = \alpha_0^a + \sum_1^n \alpha_i^a \varepsilon_i,\ \beta^a,\ \Phi^a\right),$$

where $\epsilon = (\varepsilon_1, \ldots, \varepsilon_n)^t$ belongs to Φ^a, and β^a is non negative. We use the bold face notation, $\boldsymbol{\varepsilon_i^a}$, to denote the interval concretization of ε_i. Let $\hat{a}$ and $\hat{b}$ be two constrained affine forms. Then $\hat{a} \leq \hat{b}$ in the sense of Definition 7 if and only if

$$\begin{cases} \Phi^a \subseteq \Phi^b \\ \sup_{\epsilon \in \Phi^a} |\hat{a}(\epsilon) - \hat{b}(\epsilon)| \leq \beta^b - \beta^a \end{cases}$$

In general, there is no least upper bound for two constrained affine forms, but rather, as already noted in the unconstrained case [13, 14], *minimal upper bounds*. A sufficient condition for $\hat{c}$ to be a minimal upper bound is to enforce a minimal concretization, that is, $\gamma(\hat{c}) = \gamma(\hat{a}) \cup \gamma(\hat{b})$, and then minimize β^c among upper bounds with this concretization.

Algorithm 1 computes this particular mub in some cases (when the first `return` branch is taken), and else an upper bound with minimal interval concretisation. Let us introduce the following notion used in the algorithm: let $\boldsymbol{i}$ and $\boldsymbol{j}$ be two intervals; $\boldsymbol{i}$ and $\boldsymbol{j}$ are said to be in generic position if ($\boldsymbol{i} \subseteq \boldsymbol{j}$ or $\boldsymbol{j} \subseteq \boldsymbol{i}$) imply ($\sup(\boldsymbol{i}) = \sup(\boldsymbol{j})$ or $\inf(\boldsymbol{i}) = \inf(\boldsymbol{j})$). We say by extension that two affine forms are in generic position if their interval concretizations are in generic position. The join algorithm is similar to the formula in the unconstrained case described in [14] except we have to be cautious about the relative position of the ranges of noise symbols.

Example 3. To complete the analysis of the running example of Section 1, the join of the abstract values for y on the two branches must be computed:

$$\begin{cases} \Phi^a = 1 \times [-1,1] \times [-1,1] \times [-1,1] \\ \hat{a} = 0.5 + 0.5\varepsilon_1 \\ \gamma(\hat{a}) = [0,1] \end{cases} \quad \begin{cases} \Phi^b = 1 \times [-1,-0.444] \times [-1,1] \times [-1,1] \\ \hat{b} = 14.93395 + 13.9\varepsilon_1 + 0.96605\varepsilon_3 \\ \gamma(\hat{b}) = [0.0679, 9.7284] \end{cases}$$

Algorithm 1. Join of two constrained affine forms

if $\hat{a}$ **and** $\hat{b}$ *are in generic position* **then**
  if $\mathrm{mid}(\gamma(\hat{b})) \leq \mathrm{mid}(\gamma(\hat{a}))$ **then** swap $\hat{a}$ and $\hat{b}$.
  for $i \geq 1$ **do**
    $\alpha_i^c \longleftarrow 0$
    if $\boldsymbol{\varepsilon_i^a}$ *and* $\boldsymbol{\varepsilon_i^b}$ *are in generic position* **then**
      if $\alpha_i^a \geq 0$ **and** $\alpha_i^b \geq 0$ **then**
        if $\mathrm{mid}(\boldsymbol{\varepsilon_i^a}) \leq \mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$ **and** $\mathrm{mid}(\boldsymbol{\varepsilon_i^b}) \geq \mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$ **then**
          $\alpha_i^c \longleftarrow \min(\alpha_i^a, \alpha_i^b)$
      if $\alpha_i^a \leq 0$ **and** $\alpha_i^b \leq 0$ **then**
        if $\mathrm{mid}(\boldsymbol{\varepsilon_i^a}) \geq \mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$ **and** $\mathrm{mid}(\boldsymbol{\varepsilon_i^b}) \leq \mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$ **then**
          $\alpha_i^c \longleftarrow \max(\alpha_i^a, \alpha_i^b)$
  if $0 \leq \sum_{i=1}^n \alpha_i^c(\mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b}) - \mathrm{mid}(\boldsymbol{\varepsilon_i^a})) \leq \mathrm{mid}(\gamma(\hat{a}) \cup \gamma(\hat{b})) - \mathrm{mid}(\gamma(\hat{a}))$ **and**
  $\mathrm{mid}(\gamma(\hat{a}) \cup \gamma(\hat{b})) - \mathrm{mid}(\gamma(\hat{b})) \leq \sum_{i=1}^n \alpha_i^c(\mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b}) - \mathrm{mid}(\boldsymbol{\varepsilon_i^b})) \leq 0$ **then**
    $\beta^c \longleftarrow \mathrm{dev}(\gamma(\hat{a}) \cup \gamma(\hat{b})) - \sum_{i=1}^n |\alpha_i^c| \,\mathrm{dev}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$
    $\alpha_0^c \longleftarrow \mathrm{mid}(\gamma(\hat{a}) \cup \gamma(\hat{b})) - \sum_{i=1}^n \alpha_i^c \,\mathrm{mid}(\boldsymbol{\varepsilon_i^a} \cup \boldsymbol{\varepsilon_i^b})$
    return $(\alpha_0^c, \alpha_1^c, \ldots, \alpha_n^c, \beta^c)$ /* MUB */
$\beta^c \longleftarrow \mathrm{dev}(\gamma(\hat{a}) \cup \gamma(\hat{b}))$, $\alpha_0^c \longleftarrow \mathrm{mid}(\gamma(\hat{a}) \cup \gamma(\hat{b}))$, **return** (α_0^c, β^c) /* UB */

$\hat{a}$ and $\hat{b}$ are in generic positions, and so are ε_1^a and ε_1^b, but condition $\mathrm{mid}(\boldsymbol{\varepsilon_1^b}) \geq \mathrm{mid}(\boldsymbol{\varepsilon_1^a} \cup \boldsymbol{\varepsilon_1^b})$ is not satisfied, so that the join gives the following minimal upper bound:

$$\begin{cases} \Phi^c = 1 \times [-1,1] \times [-1,1] \times [-1,1] \times [-1,1] \\ \hat{c} = 4.8642 + 4.8642\eta_1,\ \gamma(\hat{c}) = [0, 9.7284] \end{cases}$$

Example 4. Let us now consider a second example:

$$\begin{cases} \Phi^a = 1 \times [-1,0] \times [-1,1] \\ \hat{a} = 1 + 2\varepsilon_1 - \varepsilon_2,\ \gamma(\hat{a}) = [-2,2] \end{cases} \quad \begin{cases} \Phi^b = 1 \times [-1,1] \times [0,0.5] \\ \hat{b} = 4 + 3\varepsilon_1 - \varepsilon_2,\ \gamma(\hat{b}) = [-2,7] \end{cases}$$

$\hat{a}$ and $\hat{b}$ are in generic positions, as well as ε_1^a and ε_1^b, while ε_2^a and ε_2^b are not; the join gives the following minimal upper bound:

$$\begin{cases} \Phi^c = 1 \times [-1,1] \times [-1,1] \times [-1,1] \\ \hat{c} = \frac{5}{2} + 2\varepsilon_1 + \frac{5}{2}\eta_1,\ \gamma(\hat{c}) = [-2,7] \end{cases}$$

4.2 Join Operator in the General Case

As in the unconstrained case [14], mubs for the global order on *constrained affine sets* are difficult to characterize. Instead of doing so, we choose in this paper to describe a simple yet efficient way of computing a good over-approximation of such mubs, relying on Algorithm 1 for mubs with minimal concretisation for *constrained affine forms*.

We first project the constrained affine forms defining each variable of the environment (the $\pi_k(X)$, for all k) by considering all noise symbols as if they were central noise symbols. We then use Algorithm 1 to independently compute a minimal upper bound for the constrained affine form defining each variable of the environment (on $\pi_k(X)$, for all k), and introduce a new noise symbol for each variable to handle the perturbation term computed in this Algorithm. We thus obtain an upper bound of the constrained affine set.

Example 5. Consider, for all noise symbols in $[-1, 1]$, constrained affine sets X and Y defined by $x_1 = 1 + \varepsilon_1,\ x_2 = 1 + \varepsilon_2$, and $y_1 = 1 + \eta_1,\ y_2 = 1 + \eta_1$. Considering first the 1D cases, we have $x_1 \leq y_1$ and $x_2 \leq y_2$. However we do not have $X \leq Y$ for the global order of Definition 7. Applying the join operator defined here on X and Y, we construct Z, defined by $z_1 = 1 + \eta_2$ and $z_2 = 1 + \eta_3$. We now have $X \leq Z$ and $Y \leq Z$.

5 Experiments

In this section, we compare results[5] we obtain with our new domain, called constrained T1+, in its APRON implementation, with the octagon and polyhedron APRON domains and the unconstrained T1+[7]. Our constrained T1+ implementation allows to choose as a parameter of the analysis, the APRON domain we want to use to abstract the constraints on noise symbols. However, at this stage, conditionals are interpreted only for the interval domain, we thus present results for this domain only.

Table 1 shows the numerical range of a variable of interest of each test case and for each domain, after giving the exact range we would hope to find. It can be noted that on these examples, constrained T1+ is always more accurate than octagons, and is also more accurate than polyhedra on non affine problems.

Table 1. Comparison of Constrained T1+ with APRON's abstract domains

	Exact	Octagons	Polyhedra	T1+	Cons. T1+
InterQ1	$[0, 1875]$	$[-3750, 6093]$	$[-2578, 4687]$	$[0, 2500]$	$[0, 1875]$
Cosine	$[-1, 1]$	$[-1.50, 1.0]$	$[-1.50, 1.0]$	$[-1.073, 1]$	$[-1, 1]$
SinCos	$\{1\}$	$[0.84, 1.15]$	$[0.91, 1.07]$	$[0.86, 1.15]$	$[0.99, 1.00]$
InterL2	$\{0.1\}$	$[-1, 1]$	$[0.1, 0.4]$	$[-1, 1]$	$[0.1, 1]$
InterQ2	$\{0.36\}$	$[-1, 1]$	$[-0.8, 1]$	$[-1, 1]$	$[-0.4, 1]$

In Table 1, `InterQ1` combines linear tests with quadratic expressions, only constrained T1+ finds the right upper bound of the invariant. `Cosine` is a piecewise 3rd order polynomial interpolation of the cosine function: once again, only constrained T1+ finds the exact invariant. The program `SinCos` computes the sum of the squares of the sine and cosine functions (real result is 1). `InterL2` (resp. `InterQ2`) computes a piecewise affine (resp. quadratic) function of the input, then focuses on the inverse image of 1 by this function.

[5] Sources of the examples are available online
`http://www.lix.polytechnique.fr/Labo/Khalil.Ghorbal/CAV2010`

```
g(x) = sqrt(x*x-x+0.5)/sqrt(x*x+0.5);
x = [-2,2];
/* for n subdivisions */
h = 4/n;
if (-x<=h-2)
  y = g(x); z = g(y);
...
else if (-x<=i*h-2) /* i in {2,...,n-1}*/
  y = g(x); z = g(y);
...
else
  y = g(x); z = g(y);
```

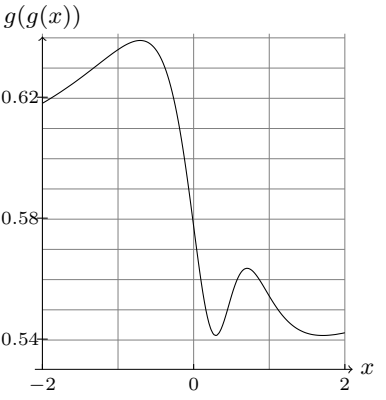


Fig. 1. Implementation of g(g(x)) for x in [-2,2] (left) and plot of g(g(x)) (right)

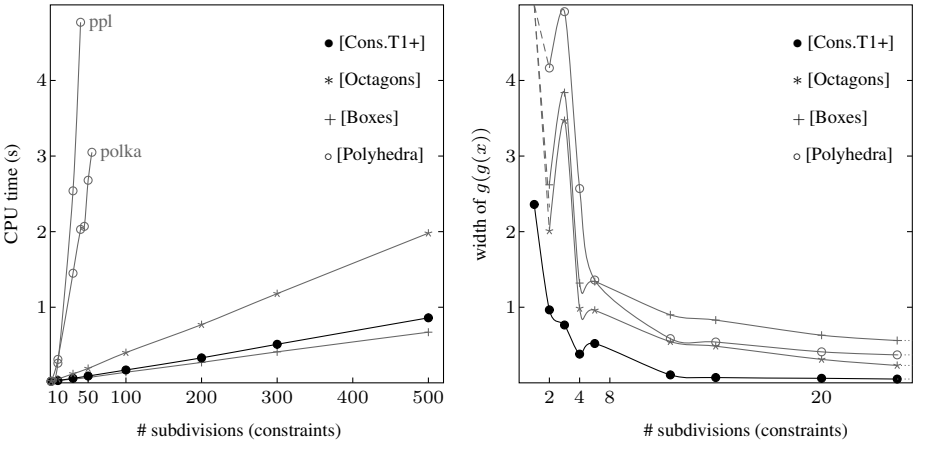


Fig. 2. Comparing analysis time and results of the different APRON domains

We now consider the computation of $g(g(x))$ on the range $x = [-2, 2]$, where

$$g(x) = \frac{\sqrt{x^2 - x + 0.5}}{\sqrt{x^2 + 0.5}}.$$

We parametrize the program that computes $g(g(x))$ by a number of tests that subdivide the domain of the input variable (see Figure 1 left for a parametrization by n subdivisions), in order to compare the relative costs and precisions of the different domains when the size of the program grows.

It can be noted (Figure 2 left) that our domain scales up well while giving here more accurate results (Figure 2 right) than the other domains. As a matter of fact, with an interval domain for the noise symbols, all abstract transfer functions are linear or at worst quadratic in the number of noise symbols appearing in the affine forms. Notice

also that our implementations detects the squares of variables, which allows constrained T1+ to give $[0, 4.72]$ without subdivisions while all other domains end with $[-\infty, +\infty]$ (noted by the dotted lines on Figure 2 right). The fact that the results observed for 3 and 5 subdivisions (Figure 2 right) are less accurate respectively than those observed for 2 and 4 subdivisions, is related to the behaviour of $g(g(x))$ on $[-2, 2]$ (see Figure 1 right): for example when a change of monotony appears near the center of a subdivision, the approximations will be less accurate than when it appears at the border.

6 Conclusion, and Future Work

In this paper, we studied the logical product of the domain of affine sets with sub-polyhedric domains on noise symbols, although the framework as described here is much more general. We concentrated on such abstract domains for $\mathcal{A}$ for practical reasons, in order to have actual algorithms to compute the abstract transfer functions.

However, in some embedded control systems, quadratic constraints appear already on the set of initial values to be treated by the control program, or as a necessary condition for behaving well, numerically speaking. For example in [3], as in a large class of navigation systems, the control program manipulates normalized quaternions, that describe the current position in 3D, of an aircraft, missile, rocket etc. We think that a combination of zonotopes with quadratic templates [1] in the lines of this article would be of interest to analyze these programs.

Also, as noticed in [2], maxplus polyhedra encompass a large subclass of disjunctions of zones; hence, by combining it with affine sets, we get another rather inexpensive way to derive a partially disjunctive analysis from affine forms (another with respect to the ideas presented in [13]).

Another future line of work is to combine the ideas of this paper with the ones of [12] to get better under-approximation methods in static analysis.

Acknowledgments. This work was partially funded by the French national research agency (ANR) projects ASOPT (Analyse Statique et OPTimisation) and Eva-Flo (Evaluation et Validation Automatique pour le Calcul Flottant) as well as by DIGITEO project PASO.

References

[1] Adje, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In: Proceedings of European Symposium on Programming (to appear, 2010)

[2] Allamigeon, X., Gaubert, S., Goubault, E.: Inferring Min and Max Invariants Using Max-plus Polyhedra. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 189–204. Springer, Heidelberg (2008)

[3] Bouissou, O., Conquet, E., Cousot, P., Cousot, R., Feret, J., Ghorbal, K., Goubault, E., Lesens, D., Mauborgne, L., Mine, A., Putot, S., Rival, X.: Space software validation using abstract interpretation. In: Proceedings of the Int. Space System Engineering Conference, Data Systems in Aerospace DASIA'09 (2009)

[4] Comba, J.L.D., Stolfi, J.: Affine arithmetic and its applications to computer graphics. In: Proceedings of SIBGRAPI (1993)
[5] Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: Proceedings of Principles Of Programming Languages, pp. 269–282. ACM Press, New York (1979)
[6] Cousot, P., Halbwachs, N.: Automatic discovery of linear restraints among variables of a program. In: Proceedings of Principles of Programming Languages, pp. 84–96. ACM Press, New York (1978)
[7] Ghorbal, K., Goubault, E., Putot, S.: The zonotope abstract domain Taylor1+. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 627–633. Springer, Heidelberg (2009)
[8] 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)
[9] Girard, A., Le Guernic, C.: Zonotope/hyperplane intersection for hybrid systems reachability analysis. In: Egerstedt, M., Mishra, B. (eds.) HSCC 2008. LNCS, vol. 4981, pp. 215–228. Springer, Heidelberg (2008)
[10] Goubault, E., Putot, S.: Weakly relational domains for floating-point computation analysis. Presented at the second international workshop on Numerical and Symbolic Abstract Domains (2005), http://www.di.ens.fr/~goubault/papers/NSAD05.pdf
[11] Goubault, E., Putot, S.: Static analysis of numerical algorithms. In: Yi, K. (ed.) SAS 2006. LNCS, vol. 4134, pp. 18–34. Springer, Heidelberg (2006)
[12] Goubault, E., Putot, S.: Under-approximations of computations in real numbers based on generalized affine arithmetic. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 137–152. Springer, Heidelberg (2007)
[13] Goubault, E., Putot, S.: Perturbed affine arithmetic for invariant computation in numerical program analysis. In: CoRR, abs/0807.2961 (2008), http://arxiv.org/abs/0807.2961
[14] Goubault, E., Putot, S.: A zonotopic framework for functional abstractions. In: CoRR, abs/0910.1763 (2009), http://arxiv.org/abs/0910.1763
[15] Gulwani, S., Tiwari, A.: Combining abstract interpreters. In: Proceedings of the ACM SIGPLAN conference on Programming language design and implementation, pp. 376–386. ACM Press, New York (2006)
[16] Keil, C.: Lurupa - rigorous error bounds in linear programming. In: Algebraic and Numerical Algorithms and Computer-assisted Proofs, Dagstuhl Seminar 5391 (2005)
[17] Laviron, V., Logozzo, F.: Subpolyhedra: A (more) scalable approach to infer linear inequalities. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 229–244. Springer, Heidelberg (2009)
[18] Manna, Z., Bradley, A.R.: The Calculus of Computation; Decision procedures with applications to verification. Springer, Heidelberg (2007)
[19] 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)
[20] APRON Project. Numerical abstract domain library (2007), http://apron.cri.ensmp.fr
[21] Sankaranarayanan, S., Sipma, H.B., Manna, Z.: Scalable analysis of linear systems using mathematical programming. In: Cousot, R. (ed.) VMCAI 2005. LNCS, vol. 3385, pp. 25–41. Springer, Heidelberg (2005)

Fast Acceleration of Ultimately Periodic Relations*

Marius Bozga[1], Radu Iosif[1], and Filip Konečný[1,2]

[1] VERIMAG, CNRS, 2 av. de Vignate, 38610 Gières, France
{bozga,iosif}@imag.fr
[2] FIT BUT, Božetěchova 2, 61266, Brno, Czech Republic
ikonecny@fit.vutbr.cz

Abstract. Computing transitive closures of integer relations is the key to finding precise invariants of integer programs. In this paper, we describe an efficient algorithm for computing the transitive closures of difference bounds, octagonal and finite monoid affine relations. On the theoretical side, this framework provides a common solution to the acceleration problem, for all these three classes of relations. In practice, according to our experiments, the new method performs up to four orders of magnitude better than the previous ones, making it a promising approach for the verification of integer programs.

1 Introduction

The verification of safety properties of infinite-state systems (such as device drivers, communication protocols, control software, etc.) requires the computation of the set of reachable states, starting with an initial state from a given (possibly infinite) set. There are currently two ways of doing this: (i) compute a finite representation of an over-approximation of the set of reachable states, by applying a widening operator at each step, or (ii) attempt to compute precisely the transitive closure of the transition relation; the set of reachable states is the image of the set of initial states via the transitive closure. The first approach is guaranteed to terminate, but the abstraction usually introduces imprecision that may blur the verification result. On the other hand, the second approach, although precise, is not guaranteed to terminate – the problem of verifying safety properties being, in general, undecidable.

In practice, one usually tries to combine the two approaches and benefit from the advantages of both. To this end, it is important to know for which classes of transition relations it is possible to compute the transitive closure precisely and fast – the relations falling outside these classes being dealt with using suitable abstractions. To the best of our knowledge, the three main classes of integer relations for which transitive closures can be computed precisely in finite time are: (1) difference bounds constraints [9,8], (2) octagons [12,6], and (3) finite monoid affine transformations [5,10]. For these three classes, the transitive closures can be moreover defined in Presburger arithmetic.

* This work was supported by the French national project ANR-09-SEGI-016 VERIDYC, by the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the internal FIT BUT grant FIT-10-1.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 227–242, 2010.

The contributions of this paper are two-fold. On the theoretical side, we show that the three classes of relations mentioned above are ultimately periodic, i.e. each relation R can be mapped into an integer matrix M_R such that the sequence $\{M_{R^k}\}_{k=0}^{\infty}$ is periodic. The proof that a sequence of matrices is ultimately periodic relies on a result from tropical semiring theory [13]. This provides shorter proofs to the fact that the transitive closures for these classes can be effectivelly computed, and that they are Presburger definable.

On the practical side, the algorithm introduced in this paper computes the transitive closure of difference bounds and octagonal relations up to four orders of magnitude faster than the original methods from [8,6], and also scales much better in the number of variables. The experimental comparison with the FAST tool [4] for difference bounds relations shows that large relations (> 50 variables), causing FAST to run out of memory, can now be handled by our implementation in less than 8 seconds, on average. We currently do not have a full implementation of the finite monoid affine transformation class, which is needed in order to compare our method with tools like FAST [4], LASH [14], or TReX [2], for this class of relations.

Related Work. Early attempts to apply Model Checking techniques to the verification of infinite-state systems consider the problem of accelerating transition relations by successive under-approximations, without any guarantee of termination. For systems with integer variables, the acceleration of affine relations has been considered primarily in the works of Annichini et. al [1], Boigelot [5], and Finkel and Leroux [10]. Finite monoid affine relations have been first studied by Boigelot [5], who shows that the finite monoid property is decidable, and that the transitive closure is Presburger definable in this case. On what concerns non-deterministic transition relations, difference bounds constraints appear in the context of timed automata verification. The transitive closure of a difference bounds constraint is shown to be Presburger definable first by Comon and Jurski [9]. Their proof was subsequently simplified and extended to parametric difference bounds constraints in [8]. We also showed that octagonal relations can be accelerated precisely, and that the transitive closure is also Presburger definable [6]. The proofs of ultimate periodicity from this paper are based on some of our previous results [8,6]. For difference bounds constraints, the proof from [8] was simplified using a result from tropical semiring theory [13].

Roadmap The paper is organized as follows: Section 2 gives the definition of ultimately periodic relations, Section 3 describes the algorithm for computing transitive closures of ultimately periodic relations, in general, Section 4 describes three instances of the algorithm, Section 5 presents the experimental results, and Section 6 concludes. Missing proofs are deferred to [7] due to reasons of space.

2 Preliminaries

We denote by $\mathbb{Z}$, $\mathbb{N}$ and $\mathbb{N}_+$ the sets of integers, positive (including zero) and strictly positive integers, respectivelly. The first order additive theory of integers is known as Presburger Arithmetic. The *tropical semiring* is defined as $\mathbb{T} = (\mathbb{Z}_\infty, \min, +, \infty, 0)$ [13], where $\mathbb{Z}_\infty = \mathbb{Z} \cup \{\infty\}$, with the extended arithmetic operations $x + \infty = \infty$, $\min(x, \infty) = x$, for all $x \in \mathbb{Z}$, where $\min(x, y)$ denotes the minimum between the

values x and y. For two square matrices $A, B \in S^{m\times m}$, we define $(A+B)_{ij} = A_{ij} + B_{ij}$ and $(A \times B)_{ij} = \min_{k=1}^{m}(a_{ik} + b_{kj})$, for all $1 \leq i, j \leq m$. Let $\mathbf{I} \in \mathbb{T}^{m\times m}$ be the identity matrix, i.e. $\mathbf{I}_{ii} = 0$ and $\mathbf{I}_{ij} = \infty$, for all $1 \leq i, j \leq m$, $i \neq j$.

Definition 1. *[13] An infinite sequence $\{s_k\}_{k=0}^{\infty} \in \mathbb{T}$ is called* ultimately periodic *if:*

$$\exists K\ \exists c > 0\ \exists \lambda_0, \lambda_1, \ldots, \lambda_{c-1} \in \mathbb{T}\ .\ s_{(k+1)c+i} = \lambda_i + s_{kc+i}$$

for all $k \geq K$ and $i = 0, 1, \ldots, c-1$. The smallest c and $\lambda_0, \lambda_1, \ldots, \lambda_{c-1}$ for which the above holds are called the period *and* rates *of $\{s_k\}_{k=0}^{\infty}$, respectivelly.*

Example 1. The sequence $\sigma_k = \{3k{+}1 \mid k = 2l,\ l \geq 2\} \cup \{5k{+}3 \mid k = 2l{+}1,\ l \geq 2\}$ is ultimately periodic, with $K = 4$, period $c = 2$ and rates $\lambda_0 = 6$, $\lambda_1 = 10$. □

A sequence of matrices $\{A_k\}_{k=0}^{\infty} \in \mathbb{T}^{m\times m}$ is said to be ultimately periodic if, for all $1 \leq i, j \leq m$, the sequence $\{(A_k)_{ij}\}_{k=0}^{\infty}$ is ultimately periodic. A matrix $A \in \mathbb{T}^{m\times m}$ is called ultimately periodic if the sequence $\{A^k\}_{k=1}^{\infty}$ is ultimately periodic, where $A^0 = \mathbf{I}$ and $A^k = A \times A^{k-1}$, for any $k > 0$. It is known that, every matrix is ultimately periodic in the tropical semiring [13].

We have the following characterization of ultimately periodic sequences of matrices:

Lemma 1. *A sequence of matrices $\{A_k\}_{k=1}^{\infty} \in \mathbb{T}^{m\times m}$ is ultimately periodic if and only if:*

$$\exists K\ \exists c > 0\ \exists \Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1} \in \mathbb{T}^{m\times m}\ .\ A_{(k+1)c+i} = \Lambda_i + A_{kc+i}$$

for all $k \geq K$ and $i = 0, 1, \ldots, c-1$.

If $A \in \mathbb{T}^{m\times m}$ is a square matrix and $n \in \mathbb{T}$, we define the matrix $(n \cdot A)_{ij} = n \cdot A_{ij}$, for all $1 \leq i, j \leq m$. If k is a parameter (typically interpreted over $\mathbb{T}$), then $\mathbb{T}[k]$ denotes the set of all terms where k may occur, built from the constants and operators of $\mathbb{T}$. For instance, if $A, B \in \mathbb{T}^{m\times m}$, then $k \cdot A + B \in \mathbb{T}[k]^{m\times m}$ denotes the matrix of terms $(k \cdot A + B)_{ij} = k \cdot A_{ij} + B_{ij}$, for all $1 \leq i, j \leq m$.

2.1 Ultimately Periodic Relations

Let $\mathbf{x} = \{x_1, x_2, \ldots, x_N\}$ be a set of variables, $N > 0$, and let $\mathbf{x}' = \{x'_1, x'_2, \ldots, x'_N\}$. A relation is an arithmetic formula $R(\mathbf{x}, \mathbf{x}')$ with free variables $\mathbf{x} \cup \mathbf{x}'$. We say that two relations R and R' are equivalent, denoted $R \Leftrightarrow R'$ if under all valuations of $\mathbf{x}$ and $\mathbf{x}'$, R is true if and only if R' is true. A relation is called *consistent* if and only if there exist valuations of $\mathbf{x}$ and $\mathbf{x}'$ under which it holds. We denote a consistent relation R by writing $R \not\Leftrightarrow$ false, and an inconsistent relation by writing $R \Leftrightarrow$ false.

The composition of two relations is defined as $R \circ R' \equiv \exists \mathbf{y}\ .\ R(\mathbf{x}, \mathbf{y}) \wedge R'(\mathbf{y}, \mathbf{x}')$. Let $\mathcal{I}$ be the identity relation $\bigwedge_{x\in\mathbf{x}} x' = x$. We define $R^0 \equiv \mathcal{I}$ and $R^n \equiv R^{n-1} \circ R$, for any $n > 0$. With these notations, $R^* \equiv \bigvee_{i=0}^{\infty} R^i$ denotes the *transitive closure* of R. A relation R is called *ω-consistent* if R^n is consistent for all $n > 0$. For the rest of this section, let $\mathcal{R}$ be a class of relations[1].

[1] A class of relations is usually defined by syntactic conditions.

Definition 2. *A relation $R(\mathbf{x}, \mathbf{x}') \in \mathcal{R}$ is called* ultimately periodic *if and only if either:*

1. *there exists $i_0 \geq 0$ such that R^{i_0} is inconsistent, or*
2. *for all $i \geq 0$, R^i is consistent, and there exists two functions:*
   - *$\sigma : \mathcal{R} \to \mathbb{T}_{\perp}^{m \times m}$ mapping each* consistent *relation in $\mathcal{R}$ into a $m \times m$ matrix of $\mathbb{T}$, for some $m > 0$, and each inconsistent relation into $\perp$.*
   - *$\rho : \mathbb{T}^{m \times m} \to \mathcal{R}$ mapping each $m \times m$ matrix of $\mathbb{T}$ into a relation in $\mathcal{R}$, such that $\rho(\sigma(R)) \Leftrightarrow R$, for each consistent relation $R \in \mathcal{R}$*

   such that the infinite sequence of matrices $\{\sigma(R^i)\}_{i=0}^{\infty} \in \mathbb{T}^{m \times m}$ is ultimately periodic.

Notice that the first condition of the definition implies that $\sigma(R^i) = \perp$, for all $i \geq i_0$. If each relation $R \in \mathcal{R}$ is ultimately periodic, then $\mathcal{R}$ is called ultimately periodic as well. The following lemma gives an alternative characterization of ω-consistent ultimately periodic relations.

Lemma 2. *An ω-consistent relation R is ultimately periodic if and only if there exist $K \geq 0$, $b \geq 0$, $c > 0$ and $\Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1} \in \mathbb{T}^{m \times m}$ such that the following hold:*

1. *$\sigma(R^{(n+1)c+i}) = \Lambda_i + \sigma(R^{nc+i})$, for all $n \geq K$.*
2. *$R^{nc+b+i} \Leftrightarrow \rho(n \cdot \Lambda_i + \sigma(R^{b+i}))$, for all $n \geq 0$.*

for all $i = 0, 1, \ldots, c-1$, where σ and ρ are the functions from Def. 2

Proof. By Lemma 1, if R is ω-consistent, then it is ultimately periodic if and only if

$$\exists K \; \exists c > 0 \; \exists \Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1} \in \mathbb{T}^{N \times N} \; . \; \sigma(R^{(k+1)c+i}) = \Lambda_i + \sigma(R^{kc+i})$$

for all $k \geq K$ and $i = 0, 1, \ldots, c-1$. By induction on $k \geq K$, one shows first that

$$R^{kc+i} \Leftrightarrow \rho({\Lambda_i}^{k-K} + \sigma(R^{Kc+i})), \; \forall k \geq K$$

Let $b = Kc$. By replacing $k - K$ with k, we obtain

$$R^{kc+b+i} \Leftrightarrow \rho({\Lambda_i}^{k} + \sigma(R^{b+i})), \; \forall k \geq 0$$

□

For practical reasons related to the representation of R^*, we are interested in finding the symbolic expression R^k, where k is a parameter (because $R^* \equiv \exists k \; . \; R^k$). Notice that the second point of lemma 2 can be used to compute the expression R^k symbolically (as a formula over $\mathbf{x}$, $\mathbf{x}'$ and k), assuming that we are given a function, call it $\pi : \mathbb{T}[k]^{m \times m} \to \mathcal{R}(k)$, where $\mathcal{R}(k)$ is the class of all parametric relations over $\mathbf{x}, \mathbf{x}'$ and k. Intuitively, π is the parametric counterpart of the ρ function from Def. 2, mapping a matrix of terms over k into a parametric relation $R(\mathbf{x}, \mathbf{x}', k)$. Concrete definitions of π will be given in Section 4.

3 Computing Transitive Closures of Ultimately Periodic Relations

In this section we give a generic algorithm that computes the transitive closure of a given ultimately periodic relation. The algorithm needs to be instantiated for a specific class $\mathcal{R}$ of ultimately periodic relations by providing the mappings σ, ρ (Def. 2) and π (the parametric counterpart of ρ) as discussed in the previous. Next, in Section 4, we show how this algorithm can be used for accelerating three classes of relations: difference bounds, octagons, and finite monoid affine transformations.

Fig. 1 shows the generic framework for computing transitive closures. The input to the algorithm is a relation R, and the mappings $\sigma : \mathcal{R} \to \mathbb{T}^{m\times m}$, $\rho : \mathbb{T}^{m\times m} \to \mathcal{R}$, and $\pi : \mathbb{T}[k]^{m\times m} \to \mathcal{R}(k)$. The algorithm is guaranteed to terminate if R is ultimately periodic, as it will be explained in the following.

The main idea of the algorithm is to discover the prefix b and period c of the sequence $\{\sigma(R^i)\}_{i=0}^{\infty}$ – cf. the second point of lemma 2. If R is ultimately periodic, such values are guaranteed to exist. The dove-tailing enumeration on lines 1 and 2 is guaranteed to yield the smallest value c for which the sequence is shown to be periodic[2].

Second, the algorithm attempts to compute the first rate of the sequence (line 6), by comparing the matrices $\sigma(R^b)$, $\sigma(R^{c+b})$ and $\sigma(R^{2c+b})$. If the difference Λ between $\sigma(R^{c+b})$ and $\sigma(R^b)$ equals the difference between $\sigma(R^{2c+b})$ and $\sigma(R^{c+b})$, then Λ is a valid candidate for the first rate of the progression (see lemma 2). Notice that the consistency check on line 4 is needed to ensure that we apply σ to consistent relations – otherwise, the relation is not ω-consistent, and the algorithm returns directly the transitive closure, i.e. the finite disjunction $\bigvee_{i=0}^{kc+b-1} R^i,\ 0 \le k \le 2$ (line 4).

Once a candidate Λ for the initial rate was found, the test $\mathcal{Q}_1$ on line 7 is used to check that R is ultimately periodic and ω-consistent. Notice that the characterization of ultimately periodic relations from lemma 2 cannot be applied here, since R^n is not known in general, for arbitrary $n \ge 0$. The condition used here is local, i.e. it needs only the relation R^b, for a typically small constant $b \ge 0$. The next lemma establishes the correctness of the criterion used by $\mathcal{Q}_1$:

Lemma 3. *An ω-consistent relation R is ultimately periodic if and only if*

$$\exists b\ \exists c > 0\ \exists \Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1} \in \mathbb{T}^{m\times m}\ .\ \rho(n \cdot \Lambda_i + \sigma(R^{b+i})) \circ R^c \Leftrightarrow \rho((n+1)\cdot \Lambda_i + \sigma(R^{b+i}))$$

for all $n \ge 0$ and $i = 0, 1, \ldots, c-1$, where σ and ρ are the functions from Def. 2. Moreover, $\Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1}$ satisfy the equivalences of Lemma 2.

Proof. "⇒" If R is ω-consistent and ultimately periodic, by Lemma 2, there exist $b \ge 0$, $c > 0$ and $\Lambda_0, \Lambda_1, \ldots, \Lambda_{c-1} \in \mathbb{T}^{m\times m}$ such that

$$R^{kc+b+i} \Leftrightarrow \rho({\Lambda_i}^k + \sigma(R^{b+i}))$$

[2] The nested loop from Fig. 1 will always yield a pair (b, c) such that $b \ge c$. To ensure that b is also minimal, and thus cover up the case $b < c$, once the smallest period c has been detected at prefix $b = c$, we need to also try all prefixes $b = c-1, c-2, \ldots, 0$.

for all $k \geq 0$ and $i = 0, 1, \ldots, c-1$. We have:

$$\begin{array}{rcl} R^{(k+1)c+b+i} & \Leftrightarrow & R^{kc+b+i} \circ R^c \\ \rho({\Lambda_i}^{k+1} + \sigma(R^{b+i})) & \Leftrightarrow & \rho({\Lambda_i}^{k} + \sigma(R^{b+i})) \circ R^c \end{array}$$

"$\Leftarrow$" We prove the equivalent condition of Lemma 2 by induction on $k \geq 0$. The base case $k = 0$ is immediate. The induction step is as follows:

$$\begin{array}{rcl} R^{(k+1)c+b+i} & \Leftrightarrow & R^{kc+b+i} \circ R^c \\ & \Leftrightarrow & \rho({\Lambda_i}^{k} + \sigma(R^{b+i})) \circ R^c \text{ , by the induction hypothesis} \\ & \Leftrightarrow & \rho({\Lambda_i}^{k+1} + \sigma(R^{b+i})) \end{array}$$

□

The universal query $\mathcal{Q}_1$ on line 7 is in general handled by procedures that are specific to the class of relations $\mathcal{R}$ we work with. Notice furthermore that $\mathcal{Q}_1$ can be handled symbolically by checking the validity of the first order formula: $\forall k \,.\, \pi(k \cdot \Lambda + \sigma(R^b)) \circ R^c \;\Leftrightarrow\; \pi((k+1) \cdot \Lambda + \sigma(R^b))$, where π is the parametric counterpart of ρ. Next, in Section 4, we detail two ways in which this test can be performed efficiently (for difference bounds and octagonal relations), without resorting to external proof engines, such as SMT or Presburger solvers.

1. foreach $b := 0, 1, 2, \ldots$ do
2. foreach $c := 1, 2, \ldots, b$ do
3. foreach $k := 0, 1, 2$ do
4. if $R^{kc+b} \Leftrightarrow$ false then return $R^* \equiv \bigvee_{i=0}^{kc+b-1} R^i$
5. endfor
6. if exists $\Lambda \in \mathbb{T}^{m \times m} : \sigma(R^{c+b}) = \Lambda + \sigma(R^b)$ and $\sigma(R^{2c+b}) = \Lambda + \sigma(R^{c+b})$ then
7. if forall $n \geq 0 : \rho(n \cdot \Lambda + \sigma(R^b)) \circ R^c \Leftrightarrow \rho((n+1) \cdot \Lambda + \sigma(R^b)) \not\Leftrightarrow$ false $(\mathcal{Q}_1)$ then
8. return $R^* \equiv \bigvee_{i=0}^{b-1} R^i \;\vee\; \exists k \geq 0 \,.\, \bigvee_{i=0}^{c-1} \pi(k \cdot \Lambda + \sigma(R^b)) \circ R^i$
9. else if exists $n \geq 0 : \rho(n \cdot \Lambda + \sigma(R^b)) \circ R^c \Leftrightarrow$ false $(\mathcal{Q}_2)$ then
10. let $n_0 = \min\{n \mid \rho(n \cdot \Lambda + \sigma(R^b)) \circ R^c \Leftrightarrow \text{false}\}$
11. if forall $n \in [0, n_0 - 1] : \rho(n \cdot \Lambda + \sigma(R^b)) \circ R^c \Leftrightarrow \rho((n+1) \cdot \Lambda + \sigma(R^b))$ then
12. return $R^* \equiv \bigvee_{i=0}^{b-1} R^i \;\vee\; \bigvee_{n=0}^{n_0-1} \bigvee_{i=0}^{c-1} \rho(n \cdot \Lambda + \sigma(R^b)) \circ R^i$
13. endif
14. endif
15. endfor
16. endfor

Fig. 1. Transitive Closure Algorithm

If the universal query on line 7 holds, the rate Λ can be used now to express the transitive closure (line 8) as a finite disjunction over the prefix $(\bigvee_{i=0}^{b-1} R^i)$ followed by a formula defining an arbitrary number of iterations $(\exists k \,.\, \bigvee_{i=0}^{c-1} \pi(k \cdot \Lambda + \sigma(R^b)) \circ R^i)$. Note that the formula on line 8 defines indeed the transitive closure of R, as a consequence of lemma 2. Moreover, this is a formula of Presburger arithmetic, provided that the classes of relations $\mathcal{R}$ and $\mathcal{R}(k)$ are Presburger definable.

Otherwise, if $\mathcal{Q}_1$ does not hold, there are two possibilities: either (i) Λ is actually not the first rate of the sequence $\{\sigma(R^i)\}_{i=0}^{\infty}$ for given $b \geq 0$ and $c > 0$, or (ii) the relation is not ω-consistent. In the first case, we need to reiterate with another prefix-period pair, which will give us another candidate Λ.

In the second case, R^m becomes inconsistent, for some $m > 0$ – in this case the computation of its transitive closure is possible, in principle, by taking the disjunction of all powers of R up to m. However, in practice this may take a long time, if m is large. In order to speed up the computation, we check whether:

- $\rho(n \cdot \Lambda + \sigma(R^b)) \circ R^c$ is inconsistent (line 9); the existential query $\mathcal{Q}_2$ (and namely finding the smallest value for which it holds) is dealt with in Section 4, specifically for the classes of difference bounds and octagonal relations.
- R is periodic with first rate Λ between 0 and $n_0 - 1$ (line 11), where n_0 is the smallest n satisfying the first point (line 10).

If both conditions above hold, then $m = (n_0 + 1)c + b$ is the smallest value for which R^m becomes inconsistent, and moreover, R is periodic with rate Λ between 0 and m. If this is the case, we compute the transitive closure using the period Λ and return (line 12). The following theorem can be proved along the lines of the discussion above:

Theorem 1. *If R is an ultimately periodic relation, the algorithm in Fig. 1 eventually terminates and returns the transitive closure of R.*

4 Some Ultimately Periodic Classes of Arithmetic Relations

This section is dedicated to the application of the transitive closure computation algorithm from the previous section (Fig. 1) to three classes of arithmetic relations, for which the transitive closure is Presburger-definable: difference bounds relations [8], octagonal relations [6], and finite monoid affine transformations [5].

In order to apply the transitive closure computation method, one needs to address two issues. First, the class of relations considered needs to be proved ultimately periodic (or else, our algorithm is not guaranteed to terminate). The proofs rely mostly on the fact that any matrix A is ultimately periodic in $\mathbb{T}$ [13] (see Section 2 for the definition of ultimately periodic matrices).

Second, the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (Fig. 1) need to be answered efficiently, by avoiding excessive calls to external decision procedures. In theory, all these queries can be expressed in Presburger arithmetic, for the classes of difference constraints, octagons and affine transformations, yet in practice we would like to avoid as much as possible using Presburger solvers, due to reasons of high complexity. For the classes of difference bounds and octagons, we give direct decision methods for handling these queries. The class of affine transformations without guards can also be dealt with by simply checking equivalence between Diophantine systems, whereas the general case still needs to be handled by a Presburger solver.

4.1 Difference Constraints

Let $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ be a set of variables ranging over $\mathbb{Z}$.

Definition 3. *A formula $\phi(\mathbf{x})$ is a* difference bounds constraint *if it is equivalent to a finite conjunction of atomic propositions of the form $x_i - x_j \leq a_{ij}$, $1 \leq i, j \leq N, i \neq j$, where $a_{ij} \in \mathbb{Z}$.*

For example, $x = y + 5$ is a difference bounds constraint, as it is equivalent to $x - y \leq 5 \wedge y - x \leq -5$. Let $\mathcal{R}_{db}$ denote the class of difference bound relations. Difference bounds constraints are alternatively represented as matrices or, equivalently, weighted graphs.

Given a difference bounds constraint ϕ, a *difference bounds matrix* (DBM) representing ϕ is a matrix $M_\phi \in \mathbb{T}^{N \times N}$ such that $(M_\phi)_{ij} = a_{ij}$, if $x_i - x_j \leq a_{ij}$ is an atomic proposition in ϕ, and ∞, otherwise. Dually, if $M \in \mathbb{T}^{N \times N}$ is a DBM, the corresponding difference bounds constraint is $\Delta_M \equiv \bigwedge_{M_{ij} < \infty} x_i - x_j \leq M_{ij}$.

A DBM M is said to be consistent if and only if its corresponding constraint φ_M is consistent. An *elementary path* in a DBM M is a sequence of indices $1 \leq i_1, i_2, \ldots, i_k \leq N$, where $i_{1,\ldots,k-1}$ are pairwise distinct, such that $M_{i_j i_{j+1}} < \infty$, for all $1 \leq j < k$. An elementary path is called an *elementary cycle* if moreover $i_1 = i_k$. An elementary cycle is said to be *strictly negative* if $\sum_{j=1}^{k-1} M_{i_j i_{j+1}} < 0$. A DBM M is inconsistent if and only if it has a strictly negative elementary cycle – a proof can be found in [12]. The next definition gives a canonical form for consistent DBMs.

Definition 4. *A consistent DBM $M \in \mathbb{T}^{N \times N}$ is said to be* closed *if and only if $M_{ii} = 0$ and $M_{ij} \leq M_{ik} + M_{kj}$, for all $1 \leq i, j, k \leq N$.*

Given a consistent DBM M, we denote by M^* the (unique) closed DBM such that $\varphi_M \Leftrightarrow \varphi_{M^*}$. It is well-known that, if M is consistent, then M^* is unique, and can be computed from M in time $\mathcal{O}(N^3)$, by the classical Floyd-Warshall algorithm. Moreover, if M is a consistent DBM, we have, for all $1 \leq i, j \leq N$:

$$M^*_{ij} = \min \left\{ \sum_{l=0}^{k-1} M_{i_l i_{l+1}} \;\middle|\; i = i_0 \ldots i_{k-1} = j \text{ is an elementary path in } M \right\} \quad (1)$$

The closed form of DBMs is needed for the elimination of existentially quantified variables – if ϕ is a difference bounds constraint, then $\exists x \,.\, \phi$ is also a difference bounds constraint [12]. Consequently, we have that the class of difference bounds relations is closed under relational composition: $R_1(\mathbf{x}, \mathbf{x}') \circ R_2(\mathbf{x}, \mathbf{x}') \equiv \exists \mathbf{y} \,.\, R_1(\mathbf{x}, \mathbf{y}) \wedge R_2(\mathbf{y}, \mathbf{x}')$.

Difference Bounds Relations are Ultimately Periodic. Given a consistent difference bounds relation $R(\mathbf{x}, \mathbf{x}') \in \mathcal{R}_{db}$, let $\sigma(R) = M_R \in \mathbb{T}^{2N \times 2N}$ be the characteristic DBM of R, and for any $M \in \mathbb{T}^{2N \times 2N}$, let $\rho(M) = \Delta_M \in \mathcal{R}_{db}$ be the difference bounds relation corresponding to R. Clearly, $\rho(\sigma(R)) \Leftrightarrow R$, as required by Def. 2.

With these definitions, the algorithm in Fig. 1 will return the transitive closure of a difference bounds relation R, provided that the sequence $\{\sigma(R^i)\}_{i=0}^{\infty}$ is ultimately periodic. If R is not ω-consistent then, by Def. 2, it is ultimately periodic. We consider henceforth that R is ω-consistent, i.e. $\sigma(R^i) = M_{R^i}$, for all $i \geq 0$.

For a difference bounds relation R, we define the directed graph $\mathcal{G}_R$, whose set of vertices is the set $\mathbf{x} \cup \mathbf{x}'$, and in which there is an edge from x_i to x_j labeled α_{ij} if and only if the atomic proposition $x_i - x_j \leq \alpha_{ij}$ occurs in R. Clearly, M_R is the incidence matrix of $\mathcal{G}_R$.

Next, we define the concatenation of $\mathcal{G}_R$ with itself as the disjoint union of two copies of $\mathcal{G}_R$, in which the $\mathbf{x}'$ vertices of the second copy overlap with the $\mathbf{x}$ vertices

of the first copy. Then R^m corresponds to the graph $\mathcal{G}_R^m$, obtained by concatenating the graph of R to itself $m > 0$ times. Since $\mathcal{R}_{db}$ is closed under relational composition, then $R^m \in \mathcal{R}_{db}$, and moreover we have:

$$\bigwedge_{1 \leq i,j \leq N} x_i - x_j \leq \min\{x_i^0 \rightarrow x_j^0\} \ \wedge \ x_i' - x_j' \leq \min\{x_i^m \rightarrow x_j^m\} \wedge$$
$$x_i - x_j' \leq \min\{x_i^0 \rightarrow x_j^m\} \ \wedge \ x_i' - x_j \leq \min\{x_i^m \rightarrow x_j^0\}$$

where $\min\{x_i^p \rightarrow x_j^q\}$ is the minimal weight of all paths between the extremal vertices x_i^p and x_j^q in $\mathcal{G}_R^m$, for $p, q \in \{0, m\}$. In other words, we have the equalities from Fig. 2 (a), for all $1 \leq i, j \leq N$.

$$\begin{array}{rcl} (M_{R^m})_{i,j} & = & \min\{x_i^0 \rightarrow x_j^0\} \\ (M_{R^m})_{i+N,j+N} & = & \min\{x_i^m \rightarrow x_j^m\} \\ (M_{R^m})_{i,j+N} & = & \min\{x_i^0 \rightarrow x_j^m\} \\ (M_{R^m})_{i+N,j} & = & \min\{x_i^m \rightarrow x_j^0\} \end{array}$$
(a)

$$\begin{array}{rcl} \min\{x_i^0 \rightarrow x_j^0\} & = & (\mathcal{M}_R^m)_{I_{ef}(x_i),F_{ef}(x_j)} \\ \min\{x_i^m \rightarrow x_j^m\} & = & (\mathcal{M}_R^m)_{I_{eb}(x_i),F_{eb}(x_j)} \\ \min\{x_i^0 \rightarrow x_j^m\} & = & (\mathcal{M}_R^m)_{I_{of}(x_i),F_{of}(x_j)} \\ \min\{x_i^m \rightarrow x_j^0\} & = & (\mathcal{M}_R^m)_{I_{ob}(x_i),F_{ob}(x_j)} \end{array}$$
(b)

Fig. 2

As proved in [8], the paths between x_i^p and x_j^q, for arbitrary $1 \leq i, j \leq N$ and $p, q \in \{0, m\}$, can be seen as words (over a finite alphabet of subgraphs of $\mathcal{G}_R^m$) recognized by a finite weighted automaton of size up to 5^N. For space reasons, the definition of this automaton is detailed in [7].

Let $\mathcal{M}_R$ be the incidence matrix of this automaton. By the construction of $\mathcal{M}_R$, for each variable $x \in \mathbf{x}$, there are eight indices, denoted as[3] $I_{of}(x), I_{ob}(x), I_{ef}(x), I_{eb}(x)$, $F_{of}(x), F_{ob}(x), F_{ef}(x), F_{eb}(x) \in \{1, \ldots, 5^N\}$, such that all relations from Fig. 2 (b) hold, for all $1 \leq i, j \leq N$. Intuitively, all paths from x_i^0 to x_j^0 are recognized by the automaton with $I_{ef}(x_i)$ and $F_{ef}(x_j)$ as the initial and final states, respectivelly. The same holds for the other pairs of indices, from Fig. 2 (b). It is easy to see (as an immediate consequence of the interpretation of the matrix product in $\mathbb{T}$) that, for any $m > 0$, the matrix $\mathcal{M}_R^m$ gives the minimal weight among all paths, of length m, between x_i^p and x_j^q, for any $1 \leq i, j \leq N$ and $p, q \in \{0, m\}$. But the sequence $\{\mathcal{M}_R^m\}_{m=0}^{\infty}$ is ultimately periodic, since every matrix is ultimately periodic in $\mathbb{T}$ [13]. By equating the relations from Fig. 2 (a) with the ones from Fig. 2 (b), we obtain that the sequence $\{\sigma(R^m)\}_{m=0}^{\infty} = \{M_{R^m}\}_{m=0}^{\infty}$ is ultimately periodic as well.

In conclusion, the algorithm from Fig. 1 will terminate on difference bounds relations. Moreover, the result is a formula in Presburger arithmetic. This also simplifies the proof that transitive closures of difference bounds relations are Presburger definable, from [8], since the minimal paths of length m within the weighted automaton recognizing the paths of $\mathcal{G}_R^m$ correspond in fact to elements of the m-th power of $\mathcal{M}_R$ (the incidence matrix of the automaton) in $\mathbb{T}$.

[3] Paths between x^0 and y^m (x^m and y^0) are called odd forward (backward) in [8], whereas paths between x^0 and y^0 (x^m and y^m) are called even forward (backward). Hence the indices of, ob, ef and eb.

Checking ω-Consistency and Inconsistency of Difference Bounds Relations. For a difference bounds relation $R(\mathbf{x}, \mathbf{x}') \in \mathcal{R}_{db}$ and a matrix $\Lambda \in \mathbb{T}^{2N \times 2N}$, we give methods to decide the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (lines 7 and 9 in Fig. 1) efficiently. To this end, we consider the class of parametric difference bounds relations. From now on, let $k \notin \mathbf{x}$ be a variable interpreted over $\mathbb{N}_+$.

Definition 5. *A formula $\phi(\mathbf{x}, k)$ is a* parametric difference bounds constraint *if it is equivalent to a finite conjunction of atomic propositions of the form $x_i - x_j \leq a_{ij} \cdot k + b_{ij}$, for some $1 \leq i, j \leq N$, $i \neq j$, where $a_{ij}, b_{ij} \in \mathbb{Z}$.*

The class of parametric difference bounds relations with parameter k is denoted as $\mathcal{R}_{db}(k)$. A parametric difference bounds constraint $\phi(k)$ can be represented by a matrix $M_\phi[k]$ of linear terms, where $(M_\phi[k])_{ij} = a_{ij} \cdot k + b_{ij}$ if $x_i - x_j \leq a_{ij} \cdot k + b_{ij}$ occurs in ϕ, and ∞ otherwise. Dually, a matrix $M[k]$ of linear terms corresponds to the formula $\Delta_M(k) \equiv \bigwedge_{M[k]_{ij} \neq \infty} x_i - x_j \leq M[k]_{ij}$. With these considerations, we define $\pi(M[k]) = \Delta_M(k)$. Clearly, we have $\pi(k \cdot \Lambda + \sigma(R^b)) \in \mathcal{R}_{db}(k)$, for $R \in \mathcal{R}_{db}$, $b \geq 0$ and $\Lambda \in \mathbb{T}^{2N \times 2N}$.

Parametric DBMs do not have a closed form, since in general, the minimum of two linear terms in k (for all valuations of k) cannot be expressed again as a linear term. According to (1), one can define the closed form of a parametric DBM as a matrix of terms of the form $\min\{a_i \cdot k + b_i\}_{i=1}^m$, for some $a_i, b_i \in \mathbb{Z}$ and $m > 0$. Then the query $\mathcal{Q}_1$ can be written as a conjunction of formulae of the form $\forall k > 0\ .\ \min\{a_i \cdot k + b_i\}_{i=1}^m = a_0 \cdot k + b_0$. The following lemma gives a way to decide the validity of such formulae:

Lemma 4. *Given $\ell, a_0, a_1, \ldots, a_m,\ b_0, b_1, \ldots, b_m \in \mathbb{Z}$, the following are equivalent:*

1. $\forall k \geq \ell\ .\ \min\{a_i \cdot k + b_i\}_{i=1}^m = a_0 \cdot k + b_0$
2. $\bigvee_{i=1}^m (a_i = a_0 \wedge b_i = b_0) \ \wedge\ \bigwedge_{j=1}^m (a_0 \leq a_j \wedge a_0 \cdot \ell + b_0 \leq a_j \cdot \ell + b_j)$

In analogy to the non-parametric case, the inconsistency of a parametric difference bounds constraint $\phi(k)$ amounts to the existence of a strictly negative elementary cycle in $M_\phi[k]$, for some valuation $k \in \mathbb{N}_+$. We are also interested in finding the smallest value for which such a cycle exists. The following lemma gives this value.

Lemma 5. *Let $\phi(\mathbf{x}, k)$ be a parametric difference bounds constraint and $M_\phi[k]$ be its associated matrix. For some $a_{ij}, b_{ij} \in \mathbb{Z}$, let $\{a_{ij} \cdot k + b_{ij}\}_{j=1}^{m_i}$, $i = 1, \ldots, 2N$ be the set of terms denoting weights of elementary cycles going through i. Then ϕ is inconsistent for some $\ell \in \mathbb{N}$ and $k \geq \ell$ if and only if there exists $1 \leq i \leq 2N$ and $1 \leq j \leq m_i$ such that either (i) $a_{ij} < 0$ or (ii) $a_{ij} \geq 0 \wedge a_{ij} \cdot \ell + b_{ij} < 0$ holds. Moreover, the smallest value for which ϕ becomes inconsistent is $\min_{i=1}^{2N}\{\min_{j=1}^{m_i} \gamma_{ij}\}$, where $\gamma_{ij} = \max(\ell, \lfloor -\frac{b_{ij}}{a_{ij}} \rfloor + 1)$, if $a_{ij} < 0$, $\gamma_{ij} = \ell$, if $a_{ij} \geq 0 \ \wedge\ a_{ij} \cdot \ell + b_{ij} < 0$, and $\gamma_{ij} = \infty$, otherwise.*

4.2 Octagons

Let $\mathbf{x} = \{x_1, x_2, ..., x_N\}$ be a set of variables ranging over $\mathbb{Z}$.

Definition 6. *A formula $\phi(\mathbf{x})$ is an* octagonal constraint *if it is equivalent to a finite conjunction of terms of the form $\pm x_i \pm x_j \leq a_{ij}$, $2x_i \leq b_i$, or $-2x_i \leq c_i$, where $a_{ij}, b_i, c_i \in \mathbb{Z}$ and $1 \leq i, j \leq N$, $i \neq j$.*

The class of octagonal relations is denoted by $\mathcal{R}_{oct}$ in the following. We represent octagons as difference bounds constraints over the set of variables $\mathbf{y} = \{y_1, y_2, \ldots, y_{2N}\}$, with the convention that y_{2i-1} stands for x_i and y_{2i} for $-x_i$, respectively. For example, the octagonal constraint $x_1 + x_2 = 3$ is represented as $y_1 - y_4 \leq 3 \wedge y_2 - y_3 \leq -3$. To handle the $\mathbf{y}$ variables in the following, we define $\bar{\imath} = i - 1$, if i is even, and $\bar{\imath} = i + 1$ if i is odd. Obviously, we have $\bar{\bar{\imath}} = i$, for all $i \in \mathbb{Z}$, $i \geq 0$. We denote by $\overline{\phi}$ the difference bounds formula $\phi[y_1/x_1, y_2/-x_1, \ldots, y_{2n-1}/x_n, y_{2n}/-x_n]$ over $\mathbf{y}$. The following equivalence relates ϕ and $\overline{\phi}$:

$$\phi(\mathbf{x}) \Leftrightarrow (\exists y_2, y_4, \ldots, y_{2N} \,.\, \overline{\phi} \wedge \bigwedge_{i=1}^{N} y_{2i-1} + y_{2i} = 0)[x_1/y_1, \ldots, x_n/y_{2N-1}] \quad (2)$$

An octagonal constraint ϕ is equivalently represented by the DBM $M_{\overline{\phi}} \in \mathbb{T}^{2N \times 2N}$, corresponding to $\overline{\phi}$. We say that a DBM $M \in \mathbb{T}^{2N \times 2N}$ is *coherent* iff $M_{ij} = M_{\bar{\jmath}\bar{\imath}}$ for all $1 \leq i, j \leq 2N$. This property is needed since any atomic proposition $x_i - x_j \leq a$, in ϕ can be represented as both $y_{2i-1} - y_{2j-1} \leq a$ and $y_{2j} - y_{2i} \leq a$, $1 \leq i, j \leq N$. Dually, a coherent DBM $M \in \mathbb{T}^{2N \times 2N}$ corresponds to the octagonal constraint Ω_M:

$$\bigwedge_{1 \leq i,j \leq N} (x_i - x_j \leq M_{2i-1,2j-1} \wedge x_i + x_j \leq M_{2i-1,2j} \wedge -x_i - x_j \leq M_{2i,2j-1}) \quad (3)$$

A coherent DBM M is said to be *octagonal-consistent* if and only if Ω_M is consistent.

Definition 7. *An octagonal-consistent coherent DBM $M \in \mathbb{T}^{2N \times 2N}$ is said to be* tightly closed *if and only if the following hold:*

1. $M_{ii} = 0$, $\forall 1 \leq i \leq 2N$
2. $M_{i\bar{\imath}}$ *is even*, $\forall 1 \leq i \leq 2N$
3. $M_{ij} \leq M_{ik} + M_{kj}$, $\forall 1 \leq i, j, k \leq 2N$
4. $M_{ij} \leq \lfloor \frac{M_{i\bar{\imath}}}{2} \rfloor + \lfloor \frac{M_{\bar{\jmath}j}}{2} \rfloor$, $\forall 1 \leq i, j \leq 2N$

The following theorem from [3] provides an effective way of testing consistency and computing the tight closure of a coherent DBM. Moreover, it shows that the tight closure of a given DBM is unique and can also be computed in time $\mathcal{O}(N^3)$.

Theorem 2. *[3] Let $M \in \mathbb{T}^{2N \times 2N}$ be a coherent DBM. Then M is octagonal-consistent if and only if M is consistent and $\lfloor \frac{M_{i\bar{\imath}}}{2} \rfloor + \lfloor \frac{M_{\bar{\jmath}j}}{2} \rfloor \geq 0$, for all $1 \leq i, j \leq 2N$, $i \neq j$. Moreover, the tight closure of M is the DBM $M^t \in \mathbb{T}^{2N \times 2N}$ defined as $M^t_{ij} = \min \left\{ M^*_{ij}, \left\lfloor \frac{M^*_{i\bar{\imath}}}{2} \right\rfloor + \left\lfloor \frac{M^*_{\bar{\jmath}j}}{2} \right\rfloor \right\}$, for all $1 \leq i, j \leq 2N$, where $M^* \in \mathbb{T}^{2N \times 2N}$ is the closure of M.*

The tight closure of an octagonal constraint is needed for existential quantifier elimination, and ultimately, for proving that the class of octagonal relations is closed under composition [6].

Octagonal Relations are Ultimately Periodic. Given a consistent octagonal relation $R(\mathbf{x}, \mathbf{x}')$ let $\sigma(R) = M_{\overline{R}}$. Dually, for any coherent DBM $M \in \mathbb{T}^{4N \times 4N}$, let $\rho(M) = \Omega_M$. Clearly, $\rho(\sigma(R)) \Leftrightarrow R$, as required by Def. 2.

In order to prove that the class $\mathcal{R}_{oct}$ of octagonal relations is ultimately periodic, we need to prove that the sequence $\{\sigma(R^m)\}_{m=0}^{\infty}$ is ultimately periodic, for an arbitrary relation $R \in \mathcal{R}_{oct}$. It is sufficient to consider only the case where R is ω-consistent, hence $\sigma(R^m) = M_{\overline{R^m}}$, for all $m \geq 0$. We rely in the following on the main result of [6], which establishes a relation between $M_{\overline{R^m}}$ (the octagonal DBM corresponding to the m-th iteration of R) and $M_{\overline{R}^m}$ (the DBM corresponding to the m-th iteration of $\overline{R} \in \mathcal{R}_{db}$), for $m > 0$:

$$(M_{\overline{R^m}})_{ij} = \min\left\{(M_{\overline{R}^m})_{ij}, \left\lfloor \frac{(M_{\overline{R}^m})_{i\bar{i}}}{2} \right\rfloor + \left\lfloor \frac{(M_{\overline{R}^m})_{\bar{j}j}}{2} \right\rfloor\right\}, \text{ for all } 1 \leq i, j \leq 4N \quad (*)$$

This relation is in fact a generalization of the tight closure expression from theorem 2, from $m = 1$ to any $m > 0$.

In Section 4.1 it was shown that difference bounds relations are ultimately periodic. In particular, this means that the sequence $\{M_{\overline{R}^m}\}_{m=0}^{\infty}$, corresponding to the iteration of the difference bounds relation $\overline{R}$, is ultimately periodic. To prove that the sequence $\{M_{\overline{R^m}}\}_{m=0}^{\infty}$ is also ultimately periodic, it is sufficient to show that: the minimum and the sum of two ultimately periodic sequences are ultimately periodic, and also that the integer half of an ultimately periodic sequence is also ultimately periodic.

Lemma 6. *Let $\{s_m\}_{m=0}^{\infty}$ and $\{t_m\}_{m=0}^{\infty}$ be two ultimately periodic sequences. Then the sequences $\{\min(s_m, t_m)\}_{m=0}^{\infty}$, $\{s_m + t_m\}_{m=0}^{\infty}$ and $\left\{\left\lfloor \frac{s_m}{2} \right\rfloor\right\}_{m=0}^{\infty}$ are ultimately periodic as well.*

Together with the above relation (*), lemma 6 proves that $\mathcal{R}_{oct}$ is ultimately periodic.

Checking ω-Consistency and Inconsistency of Octagonal Relations. This section describes an efficient method of deciding the queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (lines 7 and 9 in Fig. 1) for the class of octagonal relations. In order to deal with these queries symbolically,we need to consider first the class $\mathcal{R}_{oct}(k)$ of octagonal relations with parameter k. In the rest of this section, let $k \notin \mathbf{x}$ be a variable ranging over $\mathbb{N}_+$.

Definition 8. *Then a formula $\phi(\mathbf{x}, z)$ is a* parametric octagonal constraint *if it is equivalent to a finite conjunction of terms of the form $\pm x_i \pm x_j \leq a_{ij} \cdot k + b_{ij}$, $2x_i \leq c_i \cdot k + d_i$, or $-2x_i \leq c'_i \cdot k + d'_i$, where $a_{ij}, b_{ij}, c_i, d_i, c'_i, d'_i \in \mathbb{Z}$ and $1 \leq i, j \leq N$, $i \neq j$.*

A parametric octagon $\phi(\mathbf{x}, k)$ is represented by a matrix $M_{\phi}[k]\mathbb{T}[k]^{2N \times 2N}$ of linear terms over k, and viceversa, a matrix $M[k] \in \mathbb{T}[k]^{2N \times 2N}$ corresponds to a parametric octagon $\Omega_M(k)$. We define $\pi(M[k]) = \Omega_M(k)$. As in the case of difference bounds constraints, one notices that $\pi(k \cdot \Lambda + \sigma(R^b)) \in \mathcal{R}_{oct}(k)$, for $R \in \mathcal{R}_{oct}$, $b \geq 0$ and $\Lambda \in \mathbb{T}^{4N \times 4N}$.

The composition of parametric octagonal relations (from e.g. $\mathcal{Q}_1$) requires the computation of the tight closure in the presence of parameters. According to theorem 2, the parametric tight closure can be expressed as a matrix of elements of the form

$\min\{t_i(k)\}_{i=1}^m$, where $t_i(k)$ are either: (i) linear terms, i.e. $t_i(k) = a_i \cdot k + b_i$, or (ii) sums of halved linear terms, i.e. $t_i(k) = \lfloor \frac{a_i \cdot k + b_i}{2} \rfloor + \lfloor \frac{c_i \cdot k + d_i}{2} \rfloor$.

The main idea is to split a halved linear term of the form $\lfloor \frac{a \cdot k + b}{2} \rfloor$, $k > 0$ into two linear terms $a \cdot k + \lfloor \frac{b}{2} \rfloor$ and $a \cdot k + \lfloor \frac{b-a}{2} \rfloor$, corresponding to the cases of $k > 0$ being even or odd, respectivelly. This is justified by the following equivalence:

$$\{\lfloor \tfrac{a \cdot k + b}{2} \rfloor \mid k > 0\} = \{a \cdot k + \lfloor \tfrac{b}{2} \rfloor \mid k > 0\} \cup \{a \cdot k + \lfloor \tfrac{b-a}{2} \rfloor \mid k > 0\}$$

Hence, an expression of the form $\min\{t_i(k)\}_{i=1}^m$ yields two expressions $\min\{t_i^e(k)\}_{i=1}^m$, for even k, and $\min\{t_i^o(k)\}_{i=1}^m$, for odd k, where t_i^e and t_i^o, $1 \leq i \leq m$, are effectively computable linear terms. With these considerations, $\mathcal{Q}_1$ (for octagonal relations) is equivalent to a conjunction of equalities of the form $\forall k > 0 \,.\, \min\{t_i^\bullet(k)\}_{i=1}^m = t_0^\bullet(k)$, $\bullet \in \{e, o\}$. Now we can apply lemma 4 to the right-hand sides of the equivalences above, to give efficient equivalent conditions for deciding $\mathcal{Q}_1$.

The query $\mathcal{Q}_2$ is, according to theorem 2, equivalent to finding either (i) a strictly negative cycle in a parametric octagonal DBM $M[k]$, or (ii) a pair of indices $1 \leq i, j \leq 4N$, $i \neq j$ such that $\lfloor \frac{M[k]_{i\bar{i}}}{2} \rfloor + \lfloor \frac{M[k]_{\bar{j}j}}{2} \rfloor < 0$. Considering that the set of terms corresponding to the two cases above is $T = \{a_i \cdot k + b_i\}_{i=1}^m \cup \{\lfloor \frac{c_i \cdot k + d_i}{2} \rfloor + \lfloor \frac{e_i \cdot k + f_i}{2} \rfloor\}_{i=1}^p$, we split each term $t \in T$ into two matching linear terms, and obtain, equivalently:

$$T_{e,o} = \{\alpha_i^e \cdot k + \beta_i^e\}_{i=1}^{m+p} \cup \{\alpha_i^o \cdot k + \beta_i^o\}_{i=1}^{m+p}$$

Now we can apply lemma 5, and compute the minimal value for which a term $t \in T_{e,o}$ becomes negative, i.e. $n_0 = \min_{i=1}^{m+p} \min(2\gamma_i^e, 2\gamma_i^o - 1)$, where $\gamma_i^\bullet = \max(1, \lfloor -\frac{\beta_i^\bullet}{\alpha_i^\bullet} \rfloor + 1)$, if $\alpha_i^\bullet < 0$, 1 if $\alpha_i^\bullet \geq 0 \wedge \alpha_i^\bullet + \beta_i^\bullet < 0$, and ∞, otherwise, for $\bullet \in \{e, o\}$.

4.3 Finite Monoid Affine Transformations

The class of affine transformations is one of the most general classes of deterministic transition relations involving integer variables. If $\mathbf{x} = \langle x_1, \ldots, x_N \rangle$ is a vector of variables ranging over $\mathbb{Z}$, an *affine transformation* is a relation of the form:

$$T \equiv \mathbf{x}' = A \otimes \mathbf{x} + \mathbf{b} \wedge \phi(\mathbf{x}) \tag{4}$$

where $A \in \mathbb{Z}^{N \times N}$, $\mathbf{b} \in \mathbb{Z}^N$, ϕ is a Presburger formula, and $\otimes$ stands for the standard matrix multiplication in $\mathbb{Z}$.

The affine transformation is said to have the *finite monoid property* [5,10] if the monoid $\langle \mathcal{M}_A, \otimes \rangle$, where $\mathcal{M}_A = \{A^{\otimes^i} \mid i \geq 0\}$ is finite. In this case, we also say that A is finite monoid. Here $A^{\otimes^0} = I_N$ and $A^{\otimes^i} = A \otimes A^{\otimes^{i-1}}$, for $i > 0$. Intuitivelly, the finite monoid property is equivalent to the fact that A has finitely many powers (for the standard integer multiplication) that repeat periodically. It is easy to see that A is finite monoid if and only if there exists $p \geq 0$ and $l > 0$ such that $A^{\otimes^p} = A^{\otimes^{p+l}}$, i.e. $\mathcal{M}_A = \{A^{\otimes^0}, \ldots, A^{\otimes^p}, \ldots, A^{\otimes^{p+l-1}}\}$.

If A is finite monoid, it can be shown that T^* can be defined in Presburger arithmetic [5,10]. We achieve the same result below, by showing that finite monoid affine transformations are ultimately periodic relations. As a byproduct, the transitive closure of such relations can also be computed by the algorithm in Fig. 1.

An affine tranformation T (4) can be equivalently written in the homogeneous form:

$$T \equiv \mathbf{x'}_h = A_h \otimes \mathbf{x}_h \ \wedge \ \phi_h(\mathbf{x}_h) \text{ where } A_h \equiv \left(\begin{array}{c|c} A & \mathbf{b} \\ \hline 0 \ldots 0 & 1 \end{array}\right)$$

where $x_h = \langle x_1, \ldots x_N, x_{N+1} \rangle$ with $x_{N+1} \notin \mathbf{x}$ being a fresh variable and $\phi_h(\mathbf{x}_h) \equiv \phi(\mathbf{x}) \wedge x_{N+1} = 1$. In general, the k-th iteration of an affine transformation can be expressed as:

$$T^k \ \equiv \ \mathbf{x'}_h = {A_h}^{\otimes k} \otimes \mathbf{x}_h \ \wedge \ \forall 0 \leq \ell < k \ . \ \phi_h({A_h}^{\otimes \ell} \otimes \mathbf{x}_h) \tag{5}$$

Notice that, if $\mathbf{x}_h^{(0)}$ denotes the initial values of $\mathbf{x}_h$, the values of $\mathbf{x}_h$ at the ℓ-th iteration are $\mathbf{x}_h^{(\ell)} = {A_h}^{\otimes \ell} \otimes \mathbf{x}_h^{(0)}$. Moreover, we need to ensure that all guards up to (and including) the $(k-1)$-th step are satisfied, i.e. $\phi_h({A_h}^{\otimes \ell} \otimes \mathbf{x}_h)$, for all $0 \leq \ell < k$.

For the rest of the section we fix A and $\mathbf{b}$, as in (4). The encoding of a consistent affine transformation T is defined as $\sigma(T) = A_h \in \mathbb{T}^{(N+1)\times(N+1)}$. Dually, for some $M \in \mathbb{T}[k]^{(N+1)\times(N+1)}$, we define:

$$\pi(M) : \exists x_{N+1}, x'_{N+1} \ . \ \mathbf{x'}_h = M \otimes \mathbf{x}_h \ \wedge \ \forall 0 \leq \ell < k \ . \ \phi_h(M[\ell/k] \otimes \mathbf{x}_h)$$

where $M[\ell/k]$ denotes the matrix M in which each occurrence of k is replaced by ℓ. In contrast with the previous cases (Section 4.1 and Section 4.2), only M is not sufficient here to recover the relation $\pi(M)$ – ϕ needs to be remembered as well[4].

With these definitions, we have $\sigma(T^k) = A_h^{\otimes k}$, for all $k > 0$ – as an immediate consequence of (5). The next lemma proves that the class of finite monoid affine relations is ultimately periodic.

Lemma 7. *Given a finite monoid matrix $A \in \mathbb{Z}^{N\times N}$ and a vector $\mathbf{b} \in \mathbb{Z}^N$, the sequence $\{A_h^{\otimes k}\}_{k=0}^{\infty}$ is ultimately periodic.*

The queries $\mathcal{Q}_1$ and $\mathcal{Q}_2$ (lines 7 and 9 in Fig. 1) in the case of finite monoid affine transformations, are expressible in Presburger arithmetic. These problems could be simplified in the case of affine transformations *without guards*, i.e $T \ \equiv \ \mathbf{x}' = A\mathbf{x} + \mathbf{b}$. The transformation is, in this case, ω-consistent. Consequently, $\mathcal{Q}_1$ reduces to an equivalence between two homogeneous systems $\mathbf{x'}_h = A_{1h} \otimes \mathbf{x}_h$ and $\mathbf{x'}_h = A_{2h} \otimes \mathbf{x}_h$. This is true if and only if $A_{1h} = A_{2h}$. The query $\mathcal{Q}_2$ becomes trivially false in this case.

5 Experimental Results

We have implemented the transitive closure algorithm from Fig. 1 within the FLATA toolset [11], a framework we develop for the analysis of counter systems. We compared the performance of this algorithm with our older transitive closure computation methods for difference bounds [8] and octagonal relations [6]. We currently lack experimental data for finite monoid relations (namely, a comparison with existing tools such as FAST

[4] This incurs a slight modification of the algorithm presented in Fig. 1.

[4], LASH [14] or TReX [2] on this class), as our implementation of finite monoid affine transformation class is still underway.

Table 1 shows the results of the comparison between the older algorithms described in [8,6] (denoted as **old**) and the algorithm in Fig. 1 for difference bounds relations $d_{1,\ldots,6}$ and octagonal relations $o_{1,\ldots,6}$. The tests have been performed on both **compact** (minimum number of constraints) and **canonical** (i.e. closed, for difference bounds and tightly closed, for octagons) relations. The **speedup** column gives the ratio between the **old** and **new** execution times. The experiments were performed on a 2.53GHz machine with 2.9GB of memory.

Table 1. Comparison with older algorithms on difference bounds and octagons. Times are in milliseconds.

Relation		**new**	compact		canonical	
			old	**speedup**	**old**	**speedup**
d_0	$(x - x' = -1) \wedge (x = y')$	0.18	0.7	3.89	38.77	215.39
d_1	$(x - x' = -1) \wedge (x' = y')$	0.18	18.18	101.0	38.77	215.39
d_2	$(x - x' = -1) \wedge (x = y') \wedge (x - z' \leq 5) \wedge (z = z')$	1.2	26.5	22.1	33431.2	27859.3
d_3	$(x - x' = -1) \wedge (x = y') \wedge (x - z \leq 5) \wedge (z = z')$	0.6	32.7	54.5	33505.5	55841.7
d_4	$(x - x' = -1) \wedge (x = y) \wedge (x - z \leq 5) \wedge (z = z')$	0.5	702.3	1404.6	48913.8	97827.6
d_5	$(a = c) \wedge (b = a') \wedge (b = b') \wedge (c = c')$	1.8	5556.6	3087.0	$> 10^6$	∞
d_6	$(a - b' \leq -1) \wedge (a - e' \leq -2) \wedge (b - a' \leq -2)$ $\wedge (b - c' \leq -1) \wedge (c - b' \leq -2) \wedge (c - d' \leq -1)$ $\wedge (d - c' \leq -2) \wedge (d - e' \leq -1 \wedge e - a' \leq -1)$ $\wedge (e - d' \leq -2) \wedge (a' - b \leq 4) \wedge (a' - c \leq 3)$ $\wedge (b' - c \leq 4 \wedge b' - d \leq 3) \wedge (c' - d \leq 4) \wedge (c' - e \leq 3)$ $\wedge (d' - a \leq 3 \wedge d' - e \leq 4) \wedge (e' - a \leq 4) \wedge (e' - b \leq 3)$	5.6	$> 10^6$	∞	$> 10^6$	∞
o_1	$(x + x' = 1)$	0.21	0.91	4.33	0.91	4.33
o_2	$(x + y' \leq -1) \wedge (-y - x' \leq -2)$	0.29	0.85	2.93	0.84	2.9
o_3	$(x \leq x') \wedge (x + y' \leq -1) \wedge (-y - x' \leq -2)$	0.32	0.93	2.91	0.94	2.94
o_4	$(x + y \leq 5) \wedge (-x + x' \leq -2) \wedge (-y + y' \leq -3)$	0.21	3.67	17.48	13.52	64.38
o_5	$(x + y \leq 1) \wedge (-x \leq 0) \wedge (-y \leq 0)$	1.2	20050.9	16709.1	$> 10^6$	∞
o_6	$(x \geq 0) \wedge (y \geq 0) \wedge (x' \geq 0) \wedge (y' \geq 0)$ $\wedge (x + y \leq 1) \wedge (x' + y' \leq 1) \wedge (x - 1 \leq x')$ $\wedge (x' \leq x + 1) \wedge (y - 1 \leq y') \wedge (y' \leq y + 1)$	2.5	$> 10^6$	∞	$> 10^6$	∞

Table 2. Comparison with FAST (MONA plugin) on deterministic difference bounds. Times are in seconds. E_T – timeout 30 s, E_B – BDD too large, E_M – out of memory.

vars	FLATA			FAST				
	done	av.	E_T	done	av.	E_T	E_M	E_B
10	50	1.5	0	49	0.6	0	0	1
15	50	1.6	0	31	10.5	17	0	2
20	50	1.6	0	4	3.4	9	8	29
25	50	1.6	0	2	4.2	2	10	36
50	50	1.6	0	0	-	0	0	50
100	49	7.7	1	0	-	0	0	50

(a) – matrix density 3%

vars	FLATA			FAST				
	done	av.	E_T	done	av.	E_T	E_M	E_B
10	50	1.5	0	22	6.9	23	1	4
15	50	1.5	0	1	20.6	4	3	42
20	50	1.6	0	0	-	1	0	49
25	43	1.7	7	0	-	0	0	50
50	50	2.3	0	0	-	0	0	50
100	42	5.5	8	0	-	0	0	50

(b) – matrix density 10%

As shown in Table 1, the maximum observed speedup is almost 10^5 for difference bounds (d_4 in canonical form) and of the order of four for octagons. For the relations d_5 (canonical form), d_6 and o_6 the computation using older methods took longer than

10^6 msec. It is also worth noticing that the highest execution time with the new method was of 2.5 msec.

Table 2 compares FLATA with FAST [4] on counter systems with one self loop labeled with a randomly generated deterministic difference bound relation. We generated 50 such relations for each size $N = 10, 15, 20, 25, 50, 100$. Notice that FAST usually runs out of memory for more than 25 variables, whereas FLATA can handle 100 variables in reasonable time (less than 8 seconds on average).

6 Conclusion

We presented a new, scalable algorithm for computing the transitive closure of ultimately periodic relations. We show that this algorithm is applicable to difference bounds, octagonal and finite monoid affine relations, as all three classes are shown to be ultimately periodic. Experimental results show great improvement in the time needed to compute transitive closures of difference bounds and octagonal relations.

References

1. Annichini, A., Asarin, E., Bouajjani, A.: Symbolic techniques for parametric reasoning about counter and clock systems. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 419–434. Springer, Heidelberg (2000)
2. Annichini, A., Bouajjani, A., Sighireanu, M.: Trex: A tool for reachability analysis of complex systems. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 368–372. Springer, Heidelberg (2001)
3. Bagnara, R., Hill, P.M., Zaffanella, E.: An improved tight closure algorithm for integer octagonal constraints. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 8–21. Springer, Heidelberg (2008)
4. Bardin, S., Leroux, J., Point, G.: Fast extended release. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 63–66. Springer, Heidelberg (2006)
5. Boigelot, B.: Symbolic Methods for Exploring Infinite State Spaces, volume PhD Thesis, Vol. 189. Collection des Publications de l'Université de Liège (1999)
6. Bozga, M., Gîrlea, C., Iosif, R.: Iterating octagons. In: TACAS '09, pp. 337–351. Springer, Heidelberg (2009)
7. Bozga, M., Iosif, R., Konečńy, F.: Fast Acceleration of Ultimately Periodic Relations. Technical Report TR-2010-3, Verimag, Grenoble, France (2010)
8. Bozga, M., Iosif, R., Lakhnech, Y.: Flat parametric counter automata. Fundamenta Informaticae 91, 275–303 (2009)
9. Comon, H., Jurski, Y.: Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 268–279. Springer, Heidelberg (1998)
10. 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)
11. http://www-verinew.imag.fr/flata.html
12. Miné, A.: The octagon abstract domain. Higher-Order and Symbolic Computation 19(1), 31–100 (2006)
13. De Schutter, B.: On the ultimate behavior of the sequence of consecutive powers of a matrix in the max-plus algebra. Linear Algebra and its Applications 307, 103–117 (2000)
14. Wolper, P., Boigelot, B.: Verifying systems with infinite but regular state spaces. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 88–97. Springer, Heidelberg (1998)

An Abstraction-Refinement Approach to Verification of Artificial Neural Networks

Luca Pulina and Armando Tacchella

DIST, Università di Genova, Viale Causa, 13 – 16145 Genova, Italy
{Luca.Pulina,Armando.Tacchella}@unige.it

Abstract. A key problem in the adoption of artificial neural networks in safety-related applications is that misbehaviors can be hardly ruled out with traditional analytical or probabilistic techniques. In this paper we focus on specific networks known as Multi-Layer Perceptrons (MLPs), and we propose a solution to verify their safety using abstractions to Boolean combinations of linear arithmetic constraints. We show that our abstractions are consistent, i.e., whenever the abstract MLP is declared to be safe, the same holds for the concrete one. Spurious counterexamples, on the other hand, trigger refinements and can be leveraged to automate the correction of misbehaviors. We describe an implementation of our approach based on the HYSAT solver, detailing the abstraction-refinement process and the automated correction strategy. Finally, we present experimental results confirming the feasibility of our approach on a realistic case study.

1 Introduction

Artificial neural networks are one of the most investigated and well-established Machine Learning techniques, and they find application in a wide range of research and engineering domains – see, e.g., [1]. However, in spite of some exceptions, neural networks are confined to systems which comply only to the lowest safety integrity levels, achievable with standard industrial best practices [2]. The main reason is the absence of effective safety assurance methods for systems using neural networks. In particular, traditional analytical and probabilistic methods can be ineffective in ensuring that outputs do not generate potential hazards in safety-critical applications [3] .

In this paper we propose a formal method to verify safety of neural networks. We consider a specific kind of feed-forward neural network known as Multi-Layer Perceptron (MLP), and we state that an MLP is safe when, given every possible input value, its output is guaranteed to range within specific bounds. Even if we consider MLPs with a fairly simple topology, the *Universal Approximation Theorem* [4] guarantees that, in principle, such MLPs can approximate every non-linear real-valued function of n real-valued inputs. Also, our notion of safety is representative of all the cases in which an out-of-range response is unacceptable, such as, e.g., minimum and maximum reach of an industrial manipulator, lowest and highest percentage of a component in a mixture, and minimum and maximum dose of a drug that can be administered to a patient.

Our first contribution, in the spirit of [5], is the abstraction of MLPs to corresponding Boolean combinations of linear arithmetic constraints. Abstraction is a key enabler

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 243–257, 2010.

for verification, because MLPs are compositions of non-linear and transcendental real-valued functions, and the theories to handle such functions are undecidable [6]. Even considering rational approximations of real numbers, the amount of computational resources required to reason with realistic networks could still be prohibitive. For the MLPs that we consider, we show that our abstraction mechanism yields consistent over-approximations of concrete networks, i.e., once the abstract MLP is proven to be safe, the same holds true for the concrete one. Clearly, abstraction opens the path to spurious counterexamples, i.e., violations of the abstract safety property which fail to realize on the concrete MLP. In these cases, since we control the "coarseness" of the abstraction through a numeric parameter, it is sufficient to modify such parameter to refine the abstraction and then retry the verification. While our approach is clearly inspired by counterexample guided abstraction-refinement (CEGAR) [7], in our case refinement is not guided by the counterexample, but just caused by it, so we speak of counterexample *triggered* abstraction-refinement (CETAR).

Our second contribution is a strategy for automating MLP *repair* – a term borrowed from [8] that we use to indicate modifications of the MLP synthesis attempting to correct its misbehaviors. The idea behind repair is simple, yet fairly effective. The problem with an unsafe network is that it should be redesigned to improve its performances. This is more of an art than a science, and it has to do with various factors, including the knowledge of the physical domain in which the MLP operates. However, spurious counterexamples open an interesting path to automated repair, because they are essentially an input vector which would violate the safety constraints if the concrete MLP were to respond with less precision than what is built in it. Intuitively, since the abstract MLP consistently over-approximates the concrete one, a spurious counterexample is a weak spot of the abstract MLP which could be critical also for the concrete one. We provide strong empirical evidence in support of this intuition, and also in support of the fact that adding spurious counterexamples to the training set yields MLPs which are safer than the original ones.

We implemented the above ideas in the tool NEVER (for **Ne**ural networks **Ver**ifier) [9] which leverages HYSAT [6] to verify abstract networks and the SHARK library [10] to provide MLP infrastructure, including representation and support for evaluation and repairing. In order to test the effectiveness of our approach, we experiment with NEVER on a case study about learning the forward kinematics of an industrial manipulator. We aim to show that NEVER can handle realistic sized MLPs, as well as support the MLP designer in establishing or, at least, in improving the safety of his design in a completely automated way. The paper is structured as follows. Section 2 is a crash-course on MLPs – introducing basic notation, terminology and methodologies – and includes a detailed description of our case study. In Section 3 we describe MLP abstraction, and we prove its consistency. We also describe the basic CETAR algorithm, and we show some experiments confirming its feasibility. In Section 4, we extend the basic algorithm with automated repair, we provide empirical evidence to support the correctness of our approach, and we show experiments confirming its effectiveness in our case study. We conclude the paper in Section 5 with some final remarks and a comparison of our work with related literature.

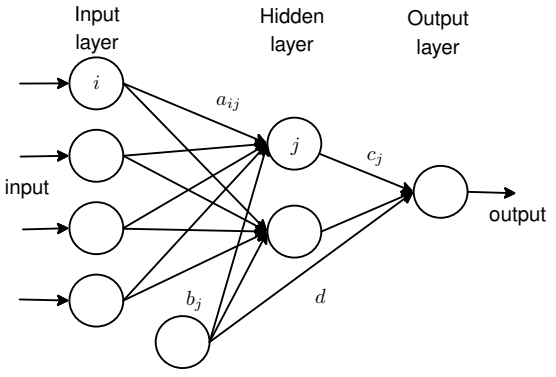


Fig. 1. Left: our MLP architecture of choice; neurons and connections are represented by circles and arrows, respectively. Right: PUMA 500 industrial manipulator.

2 Preliminaries

Structure Multi-Layer Perceptrons (MLPs) [11] are probably the most widely studied and used type of artificial neural network. An MLP is composed of a system of interconnected computing units (neurons), which are organized in layers. Figure 1 (left) shows our MLP architecture of choice, consisting of three layers: An *input layer*, that serves to pass the input vector to the network. A *hidden layer* of computation neurons. An *output layer* composed of at least a computation neuron. The MLPs that we consider are *fully connected*, i.e., each neuron is connected to every neuron in the previous and next layer. An MLP processes the information as follows. Let us consider the network ν in Figure 1. Having n neurons in the input layer ($n = 4$ in Figure 1), the i-th input value is denoted by x_i, $i = \{1, \ldots, n\}$. With m neurons in the hidden layer ($m = 2$ in Figure 1), the total input y_j received by neuron j, with $j = \{1, \ldots, m\}$, is called *induced local field* (ILF) and it is defined as

$$y_j = \sum_{i=1}^{n} a_{ji} x_i + b_j \tag{1}$$

where a_{ji} is the *weight* of the connection from the i-th neuron in the input layer to the j-th neuron in the hidden layer, and the constant b_j is the *bias* of the j-th neuron. The output of a neuron j in the hidden layer is a monotonic non-linear function of its ILF, the *activation function*. As long as such activation function is differentiable everywhere, MLPs with *only one* hidden layer can, in principle, approximate any real-valued function with n real-valued inputs [4]. A commonly used activation function [11] is the *logistic function*

$$\sigma(r) = \frac{1}{1 + \exp(-r)}, \; r \in \mathbb{R} \tag{2}$$

Therefore, the output of the MLP is

$$\nu(\underline{x}) = \sum_{j=1}^{m} c_j \sigma(y_j) + d \tag{3}$$

where c_j denotes the weight of the connection from the j-th neuron in the hidden layer to the output neuron, while d represents the bias of the output neuron. Equation (3) implies that the identity function is used as activation function of input- and output-layer neurons. This is a common choice when MLPs deal with *regression problems*. In regression problems, we are given a set of *patterns*, i.e., input vectors $X = \{\underline{x}_1, \ldots, \underline{x}_k\}$ with $\underline{x}_i \in \mathbb{R}^n$, and a corresponding set of *labels*, i.e., output values $Y = \{y_1, \ldots, y_k\}$ with $y_i \in \mathbb{R}$. We think of the labels as generated by some unknown function $f : \mathbb{R}^n \to \mathbb{R}$ applied to the patterns, i.e., $f(\underline{x}_i) = y_i$ for $i \in \{1, \ldots, k\}$. The task of ν is to *extrapolate* f given X and Y, i.e., construct ν from X and Y so that when we are given some $\underline{x}^* \notin X$ we should ensure that $\nu(\underline{x}^*)$ is "as close as possible" to $f(\underline{x}^*)$. In the following, we briefly describe how this can be achieved in practice.

Training and Validation. Given a set of patterns X and a corresponding set of labels Y generated by some unknown function f, the process of tuning the weights and the biases of an MLP ν in order to extrapolate f is called *training*, and the pair (X, Y) is called the *training set*. We can see training as a way of learning a concept, i.e., the function f, from the labelled patterns in the training set. In particular, we speak of *supervised learning* because labels can be used as a reference for training, i.e., whenever $\nu(\underline{x}_i) \neq y_i$ with $\underline{x}_i \in X$ and $y_i \in Y$ an *error signal* can be computed to determine how much the weights should be adjusted to improve the quality of the response of ν. A well-established training algorithm for MLPs is *back-propagation* (BP) [11]. Informally, an *epoch* of BP-based training is the combination of two steps. In the *forward step*, for all $i \in \{1, \ldots, k\}$, $\underline{x}_i \in X$ is input to ν, and some cumulative error measure ϵ is evaluated. In the *backward step*, the weights and the biases of the network are all adjusted in order to reduce ϵ. After a number of epochs, e.g., when ϵ stabilizes under a desired threshold, BP stops and returns the weights of the neurons, i.e., ν is the *inductive model* of f.

In general, extrapolation is an ill-posed problem. Even assuming that X and Y are sufficient to learn f, it is still the case that different sets X, Y will yield different settings of the MLP parameters. Indeed, we cannot choose elements of X and Y to guarantee that the resulting network ν will not *underfit* f, i.e., consistently deviate from f, or *overfit* f, i.e., be very close to f only when the input vector is in X. Both underfitting and overfitting lead to poor *generalization* performances, i.e., the network largely fails to predict $f(\underline{x}^*)$ on yet-to-be-seen inputs $\underline{x}^*$. Statistical techniques can provide reasonable estimates of the generalization error – see, e.g., [11]. In our experiments, we use *leave-one-out cross-validation* (or, simply, leave-one-out) which works as follows. Given the set of patterns X and the set of labels Y, we obtain the MLP $\nu_{(i)}$ by applying BP to the set of patterns $X_{(i)} = \{x_1, \ldots, x_{i-1}, x_{i+1}, \ldots x_k\}$ and to the corresponding set of labels $Y_{(i)}$. If we repeat the process k times, then we obtain k different MLPs so that we can estimate the generalization error as

$$\hat{\epsilon} = \sqrt{\frac{1}{k} \sum_{i=1}^{k} (y_i - \nu_{(i)}(\underline{x}_i))^2} \tag{4}$$

which is the root mean squared error (RMSE) among all the predictions made by each $\nu_{(i)}$ when tested on the unseen input $\underline{x_i}$. Both leave-one-out and RMSE are a common method of estimating and summarizing the generalization error in MLP applications (see e.g. [11]).

Case Study. The experiments that we present[1] concern a realistic case study about the control of a Unimate PUMA 500 industrial manipulator – see Figure 1 (right). This is a 6 degrees-of-freedom manipulator with revolute joints, which has been widely used in industry and it is still common in academic research projects. The joints are actuated by DC servo motors with encoders to locate angular positions. Our case study focuses on learning *forward kinematics*, i.e., the mapping from joint angles to end-effector position along a single coordinate of a Cartesian system having origin in the center of the robot's workspace. Our desiderata is thus to build an MLP predicting the final position of the end-effector knowing the joint angles. Since we learn the mapping using examples inside a region that we consider to be safe for the manipulator's motion, we expect the MLP to never emit a prediction that exceeds the safe region. An MLP failing to do so is to be considered unsafe. To train the MLP, we consider a training set (X, Y) collecting 141 entries. The patterns $\underline{x} \in X$ are vectors encoding the 6 joint angles, i.e., $\underline{x} = \langle \theta_1, \ldots, \theta_6 \rangle$ (in radians), and the labels are the corresponding end-effector coordinate (in meters). The range that we consider to be safe for motion goes from -0.35m to 0.35m, thus for all $y \in Y$ we have $y \in [-0.35, 0.35]$. We have built the training set using the ROBOOP library [12] which provides facilities for simulating the PUMA manipulator. The MLP was trained using the IRPROPPLUS algorithm [13], which is a modern implementation of BP. Inside our system, training an MLP to perform forward kinematics takes 0.64s across 500 epochs, yielding a RMSE estimate of the generalization error $\hat{\epsilon} = 0.024$m – the error distribution ranges from a minimum of 3.2×10^{-5}m to a maximum of 0.123m, with a median value of 0.020m. It is worth noticing that such generalization error would be considered very satisfactory in MLP applications.

3 Verifying MLPs with Abstraction

Given an MLP ν with n inputs and a single output we define

- the *input domain* of ν as a Cartesian product $\mathcal{I} = D_1 \times \ldots \times D_n$ where for all $1 \leq i \leq n$ the i-th element of the product $D_i = [a_i, b_i]$ is a closed interval bounded by $a_i, b_i \in \mathbb{R}$; and
- the *output domain* of ν as a closed interval $\mathcal{O} = [a, b]$ bounded by $a, b \in \mathbb{R}$.

In the definition above, and throughout the rest of the paper, a closed interval $[a, b]$ bounded by $a, b \in \mathbb{R}$ is the set of real numbers comprised between a and b, i.e, $[a, b] = \{x \mid a \leq x \leq b\}$ with $a \leq b$. We thus consider any MLP ν as a function $\nu : \mathcal{I} \to \mathcal{O}$, and we say that ν is *safe* if it satisfies the property

$$\forall \underline{x} \in \mathcal{I} : \nu(\underline{x}) \in [l, h] \tag{5}$$

[1] Our empirical analysis is obtained on a family of identical Linux workstations comprised of 10 Intel Core 2 Duo 2.13 GHz PCs with 4GB of RAM running Linux `Debian 2.6.18.5`.

where $l, h \in \mathcal{O}$ are *safety thresholds*, i.e., constants defining an interval wherein the MLP output is to range, given all acceptable input values. Testing exhaustively all the input vectors in $\mathcal{I}$ to make sure that ν respects condition (5) is untenable. On the other hand, statistical approaches based on sampling input vectors – see, e.g., [14] – can only give a probabilistic guarantee. In the spirit of [5], we propose to verify a *consistent abstraction* of ν, i.e., a function $\tilde{\nu}$ such that if the property corresponding to (5) is satisfied by $\tilde{\nu}$ in a suitable abstract domain, then it must hold also for ν. As in any abstraction-based approach to verification, the key point is that verifying condition (5) in the abstract domain is feasible, possibly without using excessive computational resources. This comes at the price of *spurious counterexamples*, i.e., there may exist some abstract counterexamples that do not correspond to concrete ones. A spurious counterexample calls for a refinement of the abstraction which, in turn, can make the verification process more expensive. In practice, we hope to be able to either verify ν or exhibit a counterexample within a reasonable number of refinements.

Following the framework of [5], we build abstract interpretations of MLPs where the *concrete domain* $\mathbb{R}$ is the set of real numbers, and the corresponding *abstract domain* $[\mathbb{R}] = \{[a, b] \mid a, b \in \mathbb{R}\}$ is the set of (closed) intervals of real numbers. In the abstract domain we have the usual containment relation "$\sqsubseteq$" such that given two intervals $[a, b] \in [\mathbb{R}]$ and $[c, d] \in [\mathbb{R}]$ we have that $[a, b] \sqsubseteq [c, d]$ exactly when $a \geq c$ and $b \leq d$, i.e., $[a, b]$ is a subinterval of – or it coincides with – $[c, d]$. Given any set $X \subseteq \mathbb{R}$, *abstraction* is defined as the mapping $\alpha : 2^{\mathbb{R}} \to [\mathbb{R}]$ such that

$$\alpha(X) = [\min\{X\}, \max\{X\}] \tag{6}$$

In other words, given a set $X \subseteq \mathbb{R}$, $\alpha(X)$ is the smallest interval encompassing all the elements of X, i.e., for all $x \in X$, x ranges within $\alpha(X)$ and there is no $[a, b] \sqsubseteq \alpha(X)$ for which the same holds unless $[a, b]$ coincides with $\alpha(X)$. Conversely, given $[a, b] \in [\mathbb{R}]$, *concretization* is defined as the mapping $\gamma : [\mathbb{R}] \to 2^{\mathbb{R}}$ such that

$$\gamma([a, b]) = \{x \mid x \in [a, b]\} \tag{7}$$

which represents the set of all real numbers comprised in the interval $[a, b]$. Given the posets $\langle 2^{\mathbb{R}}, \subseteq \rangle$ and $\langle [\mathbb{R}], \sqsubseteq \rangle$, the pair $\langle \alpha, \gamma \rangle$ is indeed a Galois connection because the following four properties follow from definitions (6) and (7):

1. Given two sets $X, Y \in 2^{\mathbb{R}}$, if $X \subseteq Y$ then $\alpha(X) \sqsubseteq \alpha(Y)$.
2. Given two intervals $[a, b] \in [\mathbb{R}]$ and $[c, d] \in [\mathbb{R}]$, if $[a, b] \sqsubseteq [c, d]$ then $\gamma([a, b]) \subseteq \gamma([c, d])$.
3. Given a set $X \in 2^{\mathbb{R}}$, we have that $X \subseteq \gamma(\alpha(X))$.
4. Given an interval $[a, b] \in [\mathbb{R}]$, we have that $\alpha(\gamma([a, b])$ coincides with $[a, b]$.

Let $\nu : \mathcal{I} \to \mathcal{O}$ denote the MLP for which we wish to prove safety in terms of (5). We refer to ν as the *concrete* MLP. Given a concrete domain $D = [a, b]$, the corresponding abstract domain is $[D] = \{[x, y] \mid a \leq x \leq y \leq b\}$, and we denote with $[x]$ a generic element of $[D]$. We can naturally extend the abstraction to Cartesian products of domains, i.e., given $\mathcal{I} = D_1 \times \ldots \times D_n$, we define $[\mathcal{I}] = [D_1] \times \ldots \times [D_n]$, and we denote with $[\underline{x}] = \langle [x_1], \ldots, [x_n] \rangle$ the elements of $[\mathcal{I}]$ that we call *interval vectors*.

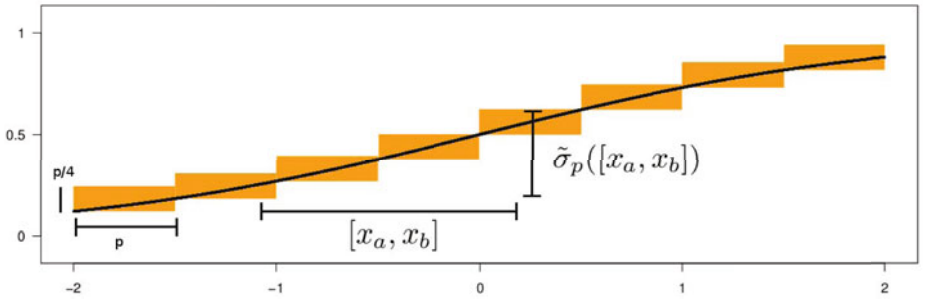


Fig. 2. Activation function $\sigma(x)$ and its abstraction $\tilde{\sigma}_p(x)$ in the range $x \in [-2, 2]$. The solid line denotes σ, while the boxes denote $\tilde{\sigma}_p$ with $p = 0.5$.

If $X \subseteq \mathcal{I}$ with $X = \{\underline{x}_1, \ldots, \underline{x}_k\}$ is a set of input vectors, then we can extend the abstraction function α by considering

$$\alpha(X) = \langle [\min_{1 \le j \le k} \{x_{1j}\}, \max_{1 \le j \le k} \{x_{1j}\}], \ldots, [\min_{1 \le i \le k} \{x_{nj}\}, \max_{1 \le j \le k} \{x_{nj}\}] \rangle \tag{8}$$

where x_{ij} denotes the i-th component ($1 \le i \le n$) of the j-th vector in X ($1 \le j \le k$). The result of $\alpha(X)$ is thus the interval vector whose components are n intervals, each obtained by considering minimum and maximum of the corresponding components in the input vectors. An *abstract* MLP $\tilde{\nu}$ is a function $\tilde{\nu} : [\mathcal{I}] \to [\mathcal{O}]$. Given a set of input vectors $X \subseteq \mathcal{I}$, $\tilde{\nu}$ provides a *consistent abstraction* of ν if it satisfies

$$\{\nu(\underline{x}) \mid \underline{x} \in X\} \subseteq \gamma(\tilde{\nu}(\alpha(X))) \tag{9}$$

In words, when given the interval vector $\alpha(X)$ as input, $\tilde{\nu}$ outputs an interval which corresponds to a superset of the values that ν would output if given as input all the vectors in X. Given our safety thresholds $l, h \in \mathcal{O}$, if we can prove

$$\forall [\underline{x}] \in [\mathcal{I}] : \tilde{\nu}([\underline{x}]) \sqsubseteq [l, h] \tag{10}$$

then, from (9) and the definition of γ, it immediately follows that

$$\{\nu(\underline{x}) \mid \underline{x} \in \mathcal{I}\} \subseteq \{y \mid l \le y \le h\} \tag{11}$$

which implies that condition (5) is satisfied by ν, because ν may not output a value outside $[l, h]$ without violating (11).

We abstract the concrete MLP ν assuming that the activation function of the hidden-layer neurons is the logistic function (2), where $\sigma(x) : \mathbb{R} \to \mathcal{O}_\sigma$ and $\mathcal{O}_\sigma = [0, 1]$. Given an abstraction parameter $p \in \mathbb{R}^+$, the *abstract activation function* $\tilde{\sigma}_p$ can be obtained by considering the maximum increment of σ over intervals of length p. Since σ is a monotonically increasing function, and its first derivative is maximum in the origin, we can use the increment of σ in the origin as the upper bound on the increment of σ elsewhere. The tangent to σ in the origin has slope $1/4$ so we have that

$$\forall x \in \mathbb{R} : 0 \le \sigma(x + p) - \sigma(x) \le \frac{p}{4} \tag{12}$$

```
NEVER(Δ, Π, [l, h], p, r)
 1  isSafe ← FALSE; isFeasible ← FALSE
 2  ν ← TRAIN(Δ, Π)
 3  repeat
 4    ν̃_p ← ABSTRACT(ν, p)
 5    s̃ ← NIL; isSafe ← CHECKSAFETY(ν̃_p, [l, h], s̃)
 6    if (not isSafe) then
 7      isFeasible ← CHECKFEASIBILITY(ν, s̃)
 8      if (not isFeasible) then
 9        p ← p / r
10  until isSafe or (not isSafe and isFeasible)
11  return isSafe
```

Fig. 3. Pseudo-code of NEVER

for any choice of the parameter $p \in \mathbb{R}^+$. Now let x_0 and x_1 be the values that satisfy $\sigma(x_0) = p/4$ and $\sigma(x_1) = 1 - p/4$, respectively. We define $\tilde{\sigma}_p : [\mathbb{R}] \to [\mathcal{O}_\sigma]$ as follows:

$$\tilde{\sigma}_p([x_a, x_b]) = \begin{cases} [0, p/4] & \text{if } x_b \leq x_0 \\ [0, \sigma(\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_a \leq x_0 \text{ and } x_b < x_1 \\ [\sigma(\lfloor \frac{x_a}{p} \rfloor), \sigma(\lfloor \frac{x_b}{p} \rfloor) + \frac{p}{4}] & \text{if } x_0 < x_a \text{ and } x_b < x_1 \\ [\sigma(\lfloor \frac{x_a}{p} \rfloor), 1] & \text{if } x_0 < x_a \text{ and } x_1 \leq x_b \\ [1 - p/4, 1] & \text{if } x_a \geq x_1 \end{cases} \tag{13}$$

Figure 2 gives a pictorial representation of the above definition. As we can see, $\tilde{\sigma}_p$ is a consistent abstraction of σ because it respects property (9) by construction. According to (13) we can control how much $\tilde{\sigma}_p$ over-approximates σ, since large values of p correspond to coarse-grained abstractions, whereas small values of p correspond to fine-grained ones. Formally, if $p < q$ then for all $[x] \in [\mathbb{R}]$, we have that $\tilde{\sigma}_p([x]) \sqsubseteq \tilde{\sigma}_q([x])$. We can now define $\tilde{\nu}_p : [\mathcal{I}] \to [\mathcal{O}]$ as

$$\tilde{\nu}_p([\underline{x}]) = \sum_{j=1}^{m} c_j \tilde{\sigma}_p(\tilde{y}_j([\underline{x}])) + d \tag{14}$$

where $\tilde{y}_j([\underline{x}]) = \sum_{i=1}^{n} a_{ji}[x_i] + b_j$, and we overload the standard symbols to denote products and sums, e.g., we write $x + y$ to mean $x \tilde{+} y$ when $x, y \in [\mathbb{R}]$. Since $\tilde{\sigma}_p$ is a consistent abstraction of σ, and products and sums on intervals are consistent abstractions of the corresponding operations on real numbers, defining $\tilde{\nu}_p$ as in (14) provides a consistent abstraction of ν. This means that our original goal of proving the safety of ν according to (5) can be now recast, modulo refinements, to the goal of proving its abstract counterpart (10).

We can leverage the above definitions to provide a complete abstraction-refinement algorithm to prove MLP safety. The pseudo-code in Figure 3 is at the core of our tool NEVER[2] which we built as proof of concept. NEVER takes as input a training set Δ, a

[2] NEVER is available for download at http://www.mind-lab.it/never. NEVER is written in C++, and it uses HYSAT to verify abstract MLPs and the SHARK library to handle representation, training, and repairing of the concrete MLPs.

Table 1. Safety checking with NEVER. The first two columns ("l" and "h") report lower and upper safety thresholds, respectively. The third column reports the final result of NEVER, and column "# CETAR" indicates the number of abstraction-refinement loops. The two columns under "TIME" report the total CPU time (in seconds) spent by NEVER and by HYSAT, respectively.

l	h	RESULT	# CETAR	TIME	
				TOTAL	HYSAT
-0.350	0.350	UNSAFE	8	1.95	1.01
-0.450	0.450	UNSAFE	9	3.15	2.10
-0.550	0.550	UNSAFE	12	6.87	5.66
-0.575	0.575	SAFE	11	6.16	4.99
-0.600	0.600	SAFE	1	0.79	0.12
-0.650	0.650	SAFE	1	0.80	0.13

set of MLP parameters Π, the safety thresholds $[l, h]$, the initial abstraction parameter p, and the refinement rate r. In line 1, two Boolean flags are defined, namely *isSafe* and *isFeasible*. The former is set to TRUE when verification of the abstract network succeeds; the latter is set to TRUE when an abstract counterexample can be realized on the concrete MLP. In line 2, a call to the function TRAIN yields a concrete MLP ν from the set Δ. The set Π must supply parameters to control topology and training of the MLP, i.e., the number of neurons in the hidden layer and the number of BP epochs. The result ν is the MLP with the least cumulative error among all the networks obtained across the epochs [10]. Lines 4 to 11 are the CETAR loop. Given p, the function ABSTRACT computes $\tilde{\nu}_p$ exactly as shown in (14) and related definitions. In line 5, CHECKSAFETY is devoted to interfacing with the HYSAT solver in order to verify $\tilde{\nu}_p$. In particular, HYSAT is supplied with a Boolean combination of linear arithmetic constraints modeling $\tilde{\nu}_p : [\mathcal{I}] \rightarrow [\mathcal{O}]$, and defining the domains $[\mathcal{I}]$ and $\mathcal{O}$, plus a further constraint encoding the safety condition. In particular, this is about finding some interval $[\underline{x}] \in [\mathcal{I}]$ such that $\tilde{\nu}([\underline{x}]) \not\sqsubseteq [l, h]$. CHECKSAFETY takes as input also a variable $\tilde{s}$ that is used to store the abstract counterexample, if any. CHECKSAFETY returns one of the following results:

- If the set of constraints supplied to HYSAT is unsatisfiable, then for all $[\underline{x}] \in [\mathcal{I}]$ we have $\tilde{\nu}_p([\underline{x}]) \sqsubseteq [l, h]$. In this case, the return value is TRUE, and $\tilde{s}$ is not set.
- If the set of constraints supplied to HYSAT is satisfiable, this means that there exists an interval $[\underline{x}] \in [\mathcal{I}]$ such that $\tilde{\nu}([\underline{x}]) \not\sqsubseteq [l, h]$. In this case, such $[\underline{x}]$ is collected in $\tilde{s}$, and the return value is FALSE.

If *isSafe* is TRUE after the call to CHECKSAFETY, then the loop ends and NEVER exits successfully. Otherwise, the abstract counterexample $\tilde{s}$ must be checked to see whether it is spurious or not. This is the task of CHECKFEASIBILITY, which takes as input the concrete MLP ν, and a concrete counterexample extracted[3] from $\tilde{s}$. If the abstract counterexample can be realized then the loop ends and NEVER exits reporting an unsuccessful verification. Otherwise, we update the abstraction parameter p according to the refinement rate r – line 9 – and we restart the loop.

We conclude this section with an experimental account of NEVER using the case study introduced in Section 2. Our main target is to find a region $[l, h]$ within which

[3] We consider a vector whose components are the midpoints of the components of the interval vector emitted by HYSAT as witness.

we can guarantee a safe calculation of the forward kinematics by means of a trained MLP. To do so, we set the initial abstraction parameter to $p = 0.5$ and the refinement rate to $r = 1.1$, and we train an MLP with 3 neurons in the hidden layer. In order to find l and h, we start by considering the interval $[-0.35, 0.35]$ – recall that this is the interval in which we consider motion to be safe. Whenever we find a counterexample stating that the network is unsafe with respect to given bounds, we enlarge the bounds. Once we have reached a safe configuration, we try to shrink the bounds, until we reach the tightest bounds that we can consider safe. The results of the above experiment are reported in Table 1. In the Table, we can see that NeVer is able to guarantee that the MLP is safe in the range $[-0.575, 0.575]$. If we try to shrink these bounds, then NeVer is always able to find a set of inputs that makes the MLP exceed the bounds. Notice that the highest total amount of CPU time corresponds to the intervals $[-0.550, 0.550]$ and $[-0.575, 0.575]$, which are the largest unsafe one and the tightest safe one, respectively. In both cases, the number of abstraction-refinement loops is also larger than other configurations that we tried.

Given that there is only one parameter governing the abstraction, we may consider whether starting with a precise abstraction, i.e., setting a relatively small value of p, would bring any advantage. However, we should keep into account that the smaller is p, the larger is the HySAT internal propositional encoding to check safety in the abstract domain. As a consequence, HySAT computations may turn out to be unfeasibly slow if the starting value of p is too small. To see this, let us consider the range $[-0.65, 0.65]$ for which Table 1 reports that HySAT solves the abstract safety check with $p = 0.5$ in 0.13 CPU seconds, and NeVer performs a single CETAR loop. The corresponding propositional encoding accounts for 599 variables and 2501 clauses in this case. If we consider the same safety check using $p = 0.05$, then we still have a single CETAR loop, but HySAT now runs for 30.26 CPU seconds, with an internal encoding of 5273 variables and 29322 clauses. Notice that the CPU time spent by HySAT in this single case is already more than the *sum* of its runtime across all the cases in Table 1. Setting $p = 0.005$ confirms this trend: HySAT solves the abstract safety check in 96116 CPU seconds (about 27 hours), and the internal encoding accounts for 50774 variables and 443400 clauses. If we consider the product between variables and clauses as a rough estimate of the encoding size, we see that a $10\times$ increase in precision corresponds to at least a $100\times$ increase in the size of the encoding. Regarding CPU times, there is more than a $200\times$ increase when going from $p = 0.5$ to $p = 0.05$, and more than a $3000\times$ increase when going from $p = 0.05$ to $p = 0.005$. In light of these results, it seems reasonable to start with coarse abstractions and let the CETAR loop refine them as needed. As we show in the following, efficiency of the automated repair heuristic is also another compelling reason behind this choice.

4 Repairing MLPs Using Spurious Counterexamples

In the previous Section we have established that, in spite of a very low generalization error, there are specific inputs to the MLP which trigger a misbehavior. As a matter of fact, the bounds in which we are able to guarantee safety would not be very satisfactory in a practical application, since they are about 64% larger than the desired ones.

This result begs the question of whether it is possible to improve MLPs response using the output of NEVER. In this section, we provide strong empirical evidence that adding spurious counterexamples to the dataset Δ and training a new MLP, yields a network whose safety bounds are tighter than the original ones. We manage to show this because our forward kinematics dataset is obtained with a simulator, so whenever a spurious counterexample is found, i.e., a vector of joint angles causing a misbehavior in the abstract network, we can compute the *true* response of the system, i.e., the position of the end-effector along a single axis. While this is feasible in our experimental setting, the problem is that MLPs are useful exactly in those cases where the target function $f : \mathcal{I} \rightarrow \mathcal{O}$ is unknown. However, we show that even in such cases the original MLP can be repaired, at least to some extent, by leveraging spurious counterexamples *and* the response of the concrete MLP under test. Intuitively, this makes sense because the concrete MLP ought to be an accurate approximation of the target function. Our experiments show that adding spurious counterexamples to the dataset Δ and training a new MLP inside the CETAR loop, also yields networks whose safety bounds are tighter than the original ones. Since Δ must contain patterns of the form $(\langle\theta_1, \ldots, \theta_6\rangle, y)$, and counterexamples are interval vectors of the form $\tilde{s} = \langle[\theta_1], \ldots, [\theta_6]\rangle$ we have the problem of determining the pattern corresponding to $\tilde{s}$ which must be added to Δ. Let ν be the MLP under test, and $\tilde{s}$ be a corresponding spurious counterexample. We proceed in two steps: First, we extract a concrete input vector $\underline{s} = \langle\theta_1, \ldots, \theta_6\rangle$ from $\tilde{s}$ as described in the previous Section. Second, we compute $\nu(\underline{s})$, and we add the pattern $(\underline{s}, \nu(\underline{s}))$ to Δ. As we can see in Figure 3, if $\tilde{s}$ is a spurious counterexample, the computation of $\underline{s}$ already comes for free because it is needed to check feasibility (line 7).

Our first experiment shows that leveraging spurious counterexamples together with their true response – a process that we call *manual-repair* in the following – yields MLPs with improved safety bounds. We consider the tightest SAFE interval in Table 1 ($[-0.575, 0.575]$), and we proceed as follows:

1. We train a new MLP ν_1 using the dataset $\Delta_1 = \Delta \cup (\underline{s}_1, f(\underline{s}_1))$ where Δ is the original dataset, $\underline{s}_1$ is extracted from $\tilde{s}$ after the first execution of the CETAR loop during the check of $[-0.575, 0.575]$, and $f(\underline{s}_1)$ is the output of the simulator.
2. We sample ten different input vectors $\{\underline{r}_1, \ldots, \underline{r}_{10}\}$, uniformly at random from the input space; for each of them, we obtain a dataset $\Gamma_i = \Delta \cup (\underline{r}_i, f(\underline{r}_i))$ where Δ and f are the same as above; finally we train ten different MLPs $\{\mu_1, \ldots, \mu_{10}\}$, where μ_i is trained on Γ_i for $1 \leq i \leq 10$.

Given the MLP ν_1 and the control MLPs $\{\mu_1, \ldots, \mu_1 0\}$, we check for their safety with NEVER. In the case of ν_1 we are able to show that the range $[-0.4, 0.4]$ is safe, which is already a considerable improvement over $[-0.575, 0.575]$. On the other hand, in the case of $\{\mu_1, \ldots, \mu_{10}\}$ the tightest bounds that we can obtain range from $[-0.47, 0.47]$ to $[-0.6, 0.6]$. This means that a targeted choice of a "weak spot" driven by a spurious counterexample turns out to be winning over a random choice. This situation is depicted in Figure 4 (left), where we can see the output of the original MLP ν corresponding to $\underline{s}_1$ (circle dot) and to $\{\underline{r}_1 \ldots \underline{r}_{10}\}$ (triangle dots). As we can see, $\nu(\underline{s}) = 0.484$ is outside the target bound of $[-0.35, 0.35]$ – notice that $f(\underline{s}) = 0.17$ in this case. On the other hand, random input vectors do not trigger, on average, an out-of-range response

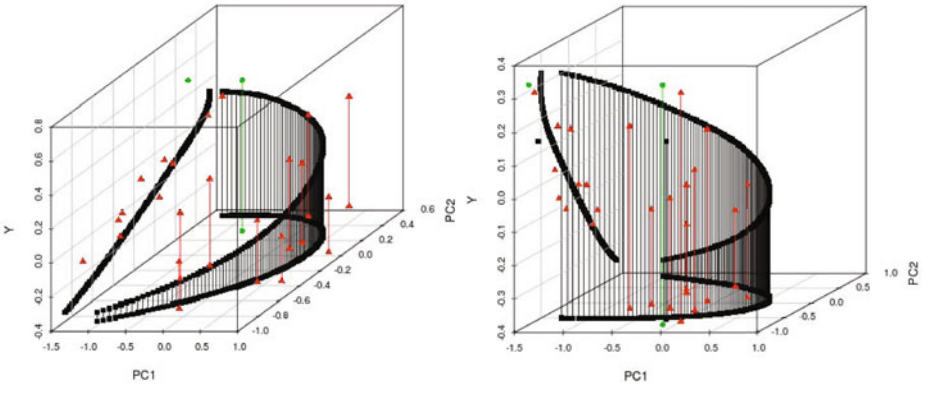


Fig. 4. Representation of ROBOOP and MLPs input-output in the manual-repair experiment. The plane (PC1-PC2) at the bottom is a two-dimensional projection of the input domain obtained considering only the first two components of a Principal Component Analysis (PCA) of the input vectors – see, e.g., Chap. 7 of [15] for an introduction to PCA. The Y axis is the output of ROBOOP and the MLPs under test. The plane (Y-PC2) on the left shows the output vs. the second principal component. All square points in space are the output of ROBOOP corresponding to the input vectors, and we also show them projected onto the (Y-PC2) plane. Circles and triangles in space are the output of the MLPs under test: circles correspond to spurious counterexamples obtained by NEVER; triangles correspond to random input samples that we use as control; for both of them we also show their projection onto the (Y-PC2) plane. For all data points, a line joins the output of the system – either ROBOOP or the MLPs under test – to the corresponding input pattern in the (PC1-PC2) plane.

of ν[4]. We repeat steps 1 and 2 above, this time considering Δ_1 as the initial dataset, and thus computing a new dataset $\Delta_2 = \Delta_1 \cup (\underline{s}_2, f(\underline{s}_2))$ where $\underline{s}_2$ is extracted from $\tilde{s}$ after the second execution of the CETAR loop. We consider a new MLP ν_2 trained on Δ_2, as well as other ten networks trained adding a random input pattern to Δ_1. Checking safety with NEVER, we are now able to show that the range $[-0.355, 0.355]$ is safe for ν_2, while the safety intervals for the remaining networks range from $[-0.4, 0.4]$ to $[-0.56, 0.56]$. In Figure 4 (right) we show graphically the results of this second round, where we can see again that the response of $\nu_1(\underline{s}_2)$ is much closer to the target bound than the response of ν_1 when considering random input patterns. In the end, the above manual-repair experiment provides strong empirical evidence that spurious counterexamples are significantly more informative than randomly chosen input patterns and that they can help in improving the original safety bounds. However, a precise theoretical explanation of the phenomenon remains to be found. In this regard, we also notice that there are cases in which training on a dataset enlarged by a single pattern may cause NEVER to be unable to confirm the same safety bounds that could be proven before. In other words, safety is not guaranteed to be preserved when adding patterns and retraining.

[4] Notice that $\underline{s}$ is still spurious in this case because we are aiming to the bound $[-0.575, 0.575]$.

Table 2. Safety checking with NEVER and repair. The table is organized as Table 1, with the only exception of column "MLP", which reports the CPU time used to train the MLP.

l	h	RESULT	# CETAR	TIME		
				TOTAL	MLP	HYSAT
-0.350	0.350	UNSAFE	11	9.50	7.31	1.65
-0.400	0.400	UNSAFE	7	6.74	4.68	1.81
-0.425	0.425	UNSAFE	13	24.93	8.74	1.52
-0.450	0.450	SAFE	3	3.11	1.92	1.10

To automate repairing, we modify NEVER by replacing lines 6-9 in the pseudo-code of Figure 3 with the following:

```
6  if (not isSafe) then
7    o ← NIL; isFeasible ← CHECKFEASIBILITY(ν, s̃, o)
8    if (not isFeasible) then
9      p ← p / r; Δ ← UPDATE(Δ, s̃, o); ν ← TRAIN(Δ, Π)
```

The parameter o is used to store the answer of ν when given $\tilde{s}$ as input. The rest of the code is meant to *update* the concrete MLP by (i) adding the input pattern extracted from the spurious counterexample $\tilde{s}$ and the corresponding output o to the set Δ, and (ii) training a new network on the extended set.

After this modification, we run a new experiment similar to the one shown in Section 3, with the aim of showing that we can improve the safety of the MLP in a completely automated, yet fairly efficient, way. Our goal is again finding values of l and h as close as possible to the ones for which the controller was trained. Table 2 shows the result of the experiment above. As we can see in the Table, we can now claim that the MLP prediction will never exceed the range $[-0.450, 0.450]$, which is "only" 28% larger than the desired one. Using this repairing heuristic in NEVER we are thus able to shrink the safety bounds of about 0.125m with respect to those obtained without repairing. This gain comes at the expense of more CPU time spent to retrain the MLP, which happens whenever we find a spurious counterexample, independently of whether NEVER will be successful in repairing the network. For instance, considering the range $[-0.350, 0.350]$ in Table 1, we see that the total CPU time spent to declare the network unsafe is 1.95s without repairing, whereas the same result with repairing takes 9.50s in Table 2. Notice that updating the MLP also implies an increase of the total amount of CETAR loops (from 8 to 11). On the other hand, still considering the range $[-0.350, 0.350]$, we can see that the average time spent by HYSAT to check the abstract network is about the same for the two cases.

Since we have shown in the previous Section that reducing p is bound to increase HYSAT runtimes substantially, automated repairing with a fixed p could be an option. Indeed, the repair procedure generates a new ν at each execution of the CETAR loop, independently from the value of p. Even if it is possible to repair the original MLP without refinement, our experiments show that this can be less effective than repair coupled with refinement. Let us consider the results reported in Table 2, and let $p = 0.5$ for each loop. We report the NEVER returns SAFE for the interval $[-0.450, 0.450]$ after 59.12s and 36 loops. The first consideration about this result concerns the CPU time spent, which is one order of magnitude higher than repair with refinement, and it is

mainly due to the higher number of retrainings. The second consideration is about the total amount of loops. Considering that the proportion of new patterns with respect to the original dataset is about 25%, and also considering that $p = 0.5$ is rather coarse, we also incur into a high risk of overfitting the MLP.

5 Conclusion and Related Work

Summing up, the abstraction-refinement approach that we proposed allows the application of formal methods to verify and repair MLPs. The two key results of our work are (i) showing that a consistent abstraction mechanism allows the verification of realistic MLPs, and (ii) showing that our repair heuristic can improve the safety of MLPs in a completely automated way. To the best of our knowledge, this is the first time in which formal verification of a functional Machine Learning technique is investigated. Contributions that are close to ours include a series of paper by Gordon, see e.g. [8], which focus on the domain of discrete-state systems with adaptive components. Since MLPs are stateless and defined over continuous variables, the results of [8] and subsequent works are unsuitable for our purposes. Robot control in the presence of safety constraints is a topic which is receiving increasing attention in recent years – see, e.g., [16]. However, the contributions in this area focus mostly on the verification of traditional, i.e., non-adaptive, methods of control. While this is a topic of interest in some fields of Machine Learning and Robotics – see, e.g., [14,3] – such contributions do not attack the problem using formal methods. Finally, since learning the weights of the connections among neurons can be viewed as synthesizing a relatively simple parametric program, our repairing procedure bears resemblances with the counterexample-driven inductive synthesis presented in [17], and the abstraction-guided synthesis presented in [18]. In both cases the setting is quite different, as the focus is on how to repair concurrent programs. However, it is probably worth investigating further connections of our work with [17,18] and, more in general, with the field of inductive programming.

References

1. Zhang, G.P.: Neural networks for classification: a survey. IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews 30(4), 451–462 (2000)
2. Smith, D.J., Simpson, K.G.L.: Functional Safety – A Straightforward Guide to applying IEC 61505 and Related Standards, 2nd edn. Elsevier, Amsterdam (2004)
3. Kurd, Z., Kelly, T., Austin, J.: Developing artificial neural networks for safety critical systems. Neural Computing & Applications 16(1), 11–19 (2007)
4. Hornik, K., Stinchcombe, M., White, H.: Multilayer feedforward networks are universal approximators. Neural networks 2(5), 359–366 (1989)
5. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages, pp. 238–252 (1977)
6. Franzle, M., Herde, C., Teige, T., Ratschan, S., Schubert, T.: Efficient solving of large nonlinear arithmetic constraint systems with complex boolean structure. Journal on Satisfiability, Boolean Modeling and Computation 1, 209–236 (2007)

7. Clarke, E., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. Journal of the ACM (JACM) 50(5), 794 (2003)
8. Gordon, D.F.: Asimovian adaptive agents. Journal of Artificial Intelligence Research 13(1), 95–153 (2000)
9. Pulina, L., Tacchella, A.: NEVER: A tool for Neural Network Verification (2010), http://www.mind-lab.it/never
10. Igel, C., Glasmachers, T., Heidrich-Meisner, V.: Shark. Journal of Machine Learning Research 9, 993–996 (2008)
11. Haykin, S.: Neural networks: a comprehensive foundation. Prentice Hall, Englewood Cliffs (2008)
12. Gordeau, R.: Roboop – a robotics object oriented package in C++ (2005), http://www.cours.polymtl.ca/roboop
13. Igel, C., Husken, M.: Empirical evaluation of the improved Rprop learning algorithms. Neurocomputing 50(1), 105–124 (2003)
14. Schumann, J., Gupta, P., Nelson, S.: On verification & validation of neural network based controllers. In: Proc. of International Conf. on Engineering Applications of Neural Networks, EANN'03 (2003)
15. Witten, I.H., Frank, E.: Data Mining, 2nd edn. Morgan Kaufmann, San Francisco (2005)
16. Pappas, G., Kress-Gazit, H. (eds.): ICRA Workshop on Formal Methods in Robotics and Automation (2009)
17. Solar-Lezama, A., Jones, C.G., Bodik, R.: Sketching concurrent data structures. In: 2008 ACM SIGPLAN conference on Programming language design and implementation, pp. 136–148. ACM, New York (2008)
18. Vechev, M., Yahav, E., Yorsh, G.G.: Abstraction-guided synthesis of synchronization. In: 37th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pp. 327–338. ACM, New York (2010)

Fences in Weak Memory Models

Jade Alglave[1], Luc Maranget[1], Susmit Sarkar[2], and Peter Sewell[2]

[1] INRIA
[2] University of Cambridge

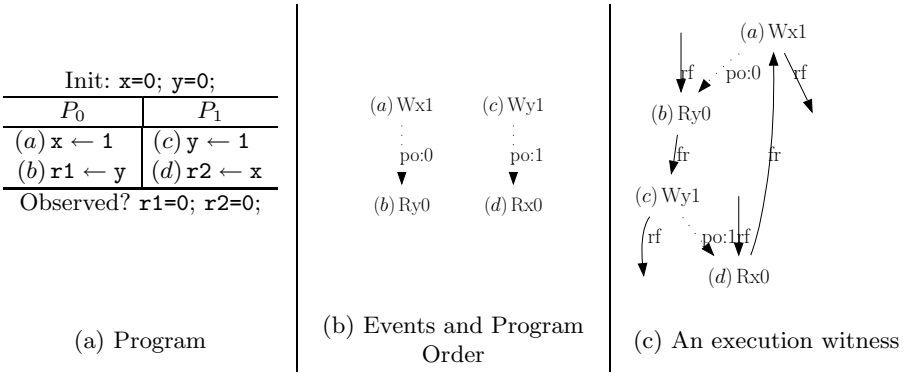
Abstract. We present a class of relaxed memory models, defined in Coq, parameterised by the chosen permitted local reorderings of reads and writes, and the visibility of inter- and intra-processor communications through memory (*e.g.* store atomicity relaxation). We prove results on the required behaviour and placement of memory fences to restore a given model (such as Sequential Consistency) from a weaker one. Based on this class of models we develop a tool, diy, that systematically and automatically generates and runs litmus tests to determine properties of processor implementations. We detail the results of our experiments on Power and the model we base on them. This work identified a rare implementation error in Power 5 memory barriers (for which IBM is providing a workaround); our results also suggest that Power 6 does not suffer from this problem.

1 Introduction

Most multiprocessors exhibit subtle relaxed-memory behaviour, with writes from one thread not immediately visible to all others; they do not provide sequentially consistent (*SC*) memory [17]. For some, such as x86 [22,20] and Power [21], the vendor documentation is in inevitably ambiguous informal prose, leading to confusion. Thus we have no foundation for software verification of concurrent systems code, and no target specification for hardware verification of microarchitecture. To remedy this state of affairs, we take a firmly empirical approach, developing, in tandem, testing tools and models of multiprocessor behaviour—the test results guiding model development and the modelling suggesting interesting tests. In this paper we make five new contributions:

1. We introduce a class of memory models, defined in Coq [8], which we show how to instantiate to produce *SC*, *TSO* [24], and a Power model (3 below).
2. We describe our diy testing tool. Much discussion of memory models has been in terms of *litmus tests* (*e.g.* **iriw** [9]): ad-hoc multiprocessor programs for which particular final states may be allowed on a given architecture. Given a violation of *SC*, diy *systematically* and *automatically* generates litmus tests (including classical ones such as **iriw**) and runs them on the hardware.
3. We model important aspects of Power processors' behaviour, *i.e. ordering relaxations*, the lack of *store atomicity* [3,7], and *A-cumulative barriers* [21].
4. We use diy to generate about 800 tests, running them up to $1e12$ times on 3 Power machines. Our experimental results confirm that our model captures many important aspects of the processor's behaviour, despite being

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 258–272, 2010.



(a) Program (b) Events and Program Order (c) An execution witness

Fig. 1. A program and a candidate execution

in a simple global-time style rather than the per-processor timelines of the architecture text. They also identified a rarely occurring implementation error in Power 5 memory barriers (for which IBM is providing a workaround). They further suggest that Power 6 does not suffer from this.

5. We prove in Coq theorems about the strength and placement of memory barriers required to regain a strong model from a weaker model.

The experimental details and the sources and documentation of diy are available online[1], as are the Coq development and typeset outlines of the proofs[2].

2 Our Class of Models

A memory model determines whether a candidate execution of a program is *valid*. For example, Fig. 1(a) shows a simple litmus test, comprising an initial state (which gathers the initial values of registers and memory locations used in the test), a program in pseudo- or assembly code, and a final condition on registers and memory (we write `x`, `y` for memory locations and `r1`, `r2` for registers). If each location initially holds 0 (henceforth we omit the initial state if so), then, *e.g.* on x86 processors, there are valid executions with the specified final state [20].

Rather than dealing directly with programs, our models are in terms of the *events* $\mathbb{E}$ occurring in a candidate program execution. A *memory event* m represents a memory access, specified by its direction (write or read), its location $\mathrm{loc}(m)$, its value $\mathrm{val}(m)$, its processor $\mathrm{proc}(m)$, and a unique label. The store to `x` with value 1 marked (a) in Fig. 1(a) generates the event (a) Wx1 in Fig. 1(b). Henceforth, we write r (resp. w) for a read (resp. write) event. We write $\mathbb{M}_{\ell,v}$ (resp. $\mathbb{R}_{\ell,v}$, $\mathbb{W}_{\ell,v}$) for the set of memory events (resp. reads, writes) to a location ℓ with value v (we omit ℓ and v when quantifying over all of them). A barrier instruction generates a *barrier event* b; we write $\mathbb{B}$ for the set of all such events.

[1] http://diy.inria.fr/
[2] http://moscova.inria.fr/~alglave/wmm/

Name	Notation	Comment	Sec.
program order	$m_1 \overset{\text{po}}{\rightarrow} m_2$	per-processor total order	2
dependencies	$m_1 \overset{\text{dp}}{\rightarrow} m_2$	dependencies	2
po-loc	$m_1 \overset{\text{po-loc}}{\rightarrow} m_2$	program order restricted to the same location	2.3
preserved program order	$m_1 \overset{\text{ppo}}{\rightarrow} m_2$	pairs maintained in program order	2.2
read-from map	$w \overset{\text{rf}}{\rightarrow} r$	links a write to a read reading its value	2.1
external read-from map	$w \overset{\text{rfe}}{\rightarrow} r$	$\overset{\text{rf}}{\rightarrow}$ between events from distinct processors	2.2
internal read-from map	$w \overset{\text{rfi}}{\rightarrow} r$	$\overset{\text{rf}}{\rightarrow}$ between events from the same processor	2.2
global read-from map	$w \overset{\text{grf}}{\rightarrow} r$	$\overset{\text{rf}}{\rightarrow}$ considered global	2.2
write serialisation	$w_1 \overset{\text{ws}}{\rightarrow} w_2$	total order on writes to the same location	2.1
from-read map	$r \overset{\text{fr}}{\rightarrow} w$	r reads from a write preceding w in $\overset{\text{ws}}{\rightarrow}$	2.1
barriers	$m_1 \overset{\text{ab}}{\rightarrow} m_2$	ordering induced by barriers	2.2
global happens-before	$m_1 \overset{\text{ghb}}{\rightarrow} m_2$	union of global relations	2.2
	$m_1 \overset{\text{hb-seq}}{\rightarrow} m_2$	shorthand for $m_1 \, (\overset{\text{rf}}{\rightarrow} \cup \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow}) \, m_2$	2.3

Fig. 2. Table of relations

The models are defined in terms of binary relations over these events, and Fig. 2 has a table of the relations we use.

As usual, the *program order* $\overset{\text{po}}{\rightarrow}$ is a total order amongst the events from the same processor that never relates events from different processors. It reflects the sequential execution of instructions on a single processor: given two instruction execution instances i_1 and i_2 that generate events e_1 and e_2, $e_1 \overset{\text{po}}{\rightarrow} e_2$ means that a sequential processor would execute i_1 before i_2. When instructions may perform several memory accesses, we take intra-instruction dependencies [22] into account to build a total order.

We postulate a $\overset{\text{dp}}{\rightarrow}$ relation to model the dependencies between instructions, such as *data* or *control dependencies* [21, pp. 653-668]. This relation is a subrelation of $\overset{\text{po}}{\rightarrow}$, and always has a read as its source.

2.1 Execution Witnesses

Although $\overset{\text{po}}{\rightarrow}$ conveys important features of program execution, *e.g.* branch resolution, it does not characterise an execution. To do so, we postulate two relations $\overset{\text{rf}}{\rightarrow}$ and $\overset{\text{ws}}{\rightarrow}$ over memory events.

Reads-from map. We write $w \overset{\text{rf}}{\rightarrow} r$ to mean that r loads the value stored by w (so w and r must share the same location and value). Given a read r there exists a unique write w such that $w \overset{\text{rf}}{\rightarrow} r$ (w can be an *init* store when r loads from the initial state). Thus, $\overset{\text{rf}}{\rightarrow}$ must be well formed following the $\text{wf} - \text{rf}$ predicate:

$$\text{wf} - \text{rf}(\overset{\text{rf}}{\rightarrow}) \triangleq \left(\overset{\text{rf}}{\rightarrow} \subseteq \bigcup_{\ell,v} (\mathbb{W}_{\ell,v} \times \mathbb{R}_{\ell,v}) \right) \wedge (\forall r, \exists! \, w. \, w \overset{\text{rf}}{\rightarrow} r)$$

Write serialisation. We assume all values written to a given location ℓ to be serialised, following a *coherence order*. This property is widely assumed by modern architectures. We define $\overset{\text{ws}}{\rightarrow}$ as the union of the coherence orders for all memory locations, which must be well formed following the wf − ws predicate:

$$\text{wf} - \text{ws}(\overset{\text{ws}}{\rightarrow}) \triangleq \left(\overset{\text{ws}}{\rightarrow} \subseteq \bigcup_{\ell} (\mathbb{W}_\ell \times \mathbb{W}_\ell)\right) \wedge \left(\forall \ell.\ \text{total} - \text{order}\left(\overset{\text{ws}}{\rightarrow}, (\mathbb{W}_\ell \times \mathbb{W}_\ell)\right)\right)$$

From-read map. We define the following derived relation $\overset{\text{fr}}{\rightarrow}$ [4] which gathers all pairs of reads r and writes w such that r reads from a write that is before w in $\overset{\text{ws}}{\rightarrow}$:

$$r \overset{\text{fr}}{\rightarrow} w \triangleq \exists\, w'.\, w' \overset{\text{rf}}{\rightarrow} r \wedge w' \overset{\text{ws}}{\rightarrow} w$$

We define an *execution witness* X as follows (the well-formedness predicate wf on execution witnesses is the conjunction of those for $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{rf}}{\rightarrow}$):

$$X \triangleq (\mathbb{E}, \overset{\text{po}}{\rightarrow}, \overset{\text{dp}}{\rightarrow}, \overset{\text{rf}}{\rightarrow}, \overset{\text{ws}}{\rightarrow})$$

Fig. 1(c) shows an execution witness for the test of Fig. 1(a). The load (d) reads the initial value of `x`, later overwritten by the store (a). Since the init store to `x` comes first in $\overset{\text{ws}}{\rightarrow}$, hence before (a), we have $(d) \overset{\text{fr}}{\rightarrow} (a)$.

2.2 Global Happens-Before

An execution witness is valid if the memory events can be embedded in an acyclic *global happens-before* relation $\overset{\text{ghb}}{\rightarrow}$ (together with two auxiliary conditions detailed in Sec. 2.3). This order corresponds roughly to the vendor documentation concept of memory events being *globally performed* [21,13]: a write in $\overset{\text{ghb}}{\rightarrow}$ represents the point in global time when this write becomes visible to all processors; whereas a read in $\overset{\text{ghb}}{\rightarrow}$ represents the point in global time when the read takes place.

There remain key choices as to which relations we include in $\overset{\text{ghb}}{\rightarrow}$ (*i.e.* which we consider to be in global time), which leads us to define a class of models.

Globality. Writes are not necessarily globally performed at once. Thus, $\overset{\text{rf}}{\rightarrow}$ is not necessarily included in $\overset{\text{ghb}}{\rightarrow}$. Let us distinguish between internal (resp. external) $\overset{\text{rf}}{\rightarrow}$, when the two events in $\overset{\text{rf}}{\rightarrow}$ are on the same (resp. distinct) processor(s), written $\overset{\text{rfi}}{\rightarrow}$ (resp. $\overset{\text{rfe}}{\rightarrow}$) : $w \overset{\text{rfi}}{\rightarrow} r \triangleq w \overset{\text{rf}}{\rightarrow} r \wedge \text{proc}(w) = \text{proc}(r)$ and $w \overset{\text{rfe}}{\rightarrow} r \triangleq w \overset{\text{rf}}{\rightarrow} r \wedge \text{proc}(w) \neq \text{proc}(r)$. Some architectures allow *store forwarding* (or *read own writes early* [3]): the processor issuing a given write can read its value before any other participant accesses it. Then $\overset{\text{rfi}}{\rightarrow}$ is not included in $\overset{\text{ghb}}{\rightarrow}$. Other architectures allow two processors sharing a cache to read a write issued by their neighbour *w.r.t.* the cache hierarchy before any other participant that does not share the same cache—a particular case of *read others' writes early* [3]. Then $\overset{\text{rfe}}{\rightarrow}$ is not considered global. We write $\overset{\text{grf}}{\rightarrow}$ for the subrelation of $\overset{\text{rf}}{\rightarrow}$ included in $\overset{\text{ghb}}{\rightarrow}$.

In our class of models, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ are always included in $\overset{\text{ghb}}{\rightarrow}$. Indeed, the write serialisation for a given location ℓ is the order in which writes to ℓ are globally performed. Moreover, as $r \overset{\text{fr}}{\rightarrow} w$ expresses that the write w' from which r reads is globally performed before w, it forces the read r to be globally performed (since a read is globally performed as soon as it is performed) before w is globally performed.

Preserved program order. In any given architecture, certain pairs of events in the program order are guaranteed to occur in that order. We postulate a global relation $\overset{\text{ppo}}{\rightarrow}$ gathering all such pairs. For example, the execution witness in Fig. 1(c) is only valid if the writes and reads to different locations on each processor have been reordered. Indeed, if these pairs were forced to be in program order, we would have a cycle in $\overset{\text{ghb}}{\rightarrow}$: $(a) \overset{\text{ppo}}{\rightarrow} (b) \overset{\text{fr}}{\rightarrow} (c) \overset{\text{ppo}}{\rightarrow} (d) \overset{\text{fr}}{\rightarrow} (a)$.

Barrier constraints. Architectures also provide *barrier* instructions, *e.g.* the Power `sync` (discussed in Sec. 3) to enforce ordering between pairs of events. We postulate a global relation $\overset{\text{ab}}{\rightarrow}$ gathering all such pairs.

Architectures. We call a particular model of our class an *architecture*, written A (or A^{ϵ} for when $\overset{\text{ab}}{\rightarrow}$ is empty); *ppo* (resp. *grf*, *ab*, A.ghb) is the function returning the $\overset{\text{ppo}}{\rightarrow}$ (resp. $\overset{\text{grf}}{\rightarrow}$, $\overset{\text{ab}}{\rightarrow}$ and $\overset{\text{ghb}}{\rightarrow}$) relation when given an execution witness:

$$A \triangleq (ppo, grf, ab)$$

We define $\overset{\text{ghb}}{\rightarrow}$ as the union of the global relations:

$$\overset{\text{ghb}}{\rightarrow} \triangleq \overset{\text{ppo}}{\rightarrow} \cup \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow} \cup \overset{\text{grf}}{\rightarrow} \cup \overset{\text{ab}}{\rightarrow}$$

2.3 Validity of an Execution *w.r.t.* an Architecture

We now add two sanity conditions to the above. First, we require each processor to respect memory coherence for each location [11]. If a processor writes *e.g.* v to ℓ and then reads v' from ℓ, v' should not precede v in the write serialisation. We define the relation $\overset{\text{po-loc}}{\rightarrow}$ over accesses to the same location in the program order, and require $\overset{\text{po-loc}}{\rightarrow}$, $\overset{\text{rf}}{\rightarrow}$, $\overset{\text{ws}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ to be compatible (writing $\overset{\text{hb-seq}}{\rightarrow}$ for $\overset{\text{rf}}{\rightarrow} \cup \overset{\text{ws}}{\rightarrow} \cup \overset{\text{fr}}{\rightarrow}$):

$$m_1 \overset{\text{po-loc}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{po}}{\rightarrow} m_2 \wedge \text{loc}(m_1) = \text{loc}(m_2)$$
$$\text{uniproc}(X) \triangleq \text{acyclic}(\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po-loc}}{\rightarrow})$$

For example, in Fig. 3 (a), we have $(c) \overset{\text{ws}}{\rightarrow} (a)$ (by `x` final value) and $(a) \overset{\text{rf}}{\rightarrow} (c)$ (by `r1` final value). The cycle $(a) \overset{\text{rf}}{\rightarrow} (b) \overset{\text{po-loc}}{\rightarrow} (c) \overset{\text{ws}}{\rightarrow} (a)$ invalidates this execution: (b) cannot read from (a) as it is a future value of `x` in $\overset{\text{ws}}{\rightarrow}$.

Second, we rule out programs where values come *out of thin air* [19] (as in Fig. 3 (b)):

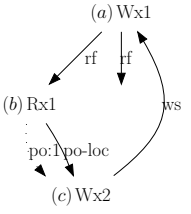
$$\text{thin}(X) \triangleq \text{acyclic}(\overset{\text{rf}}{\rightarrow} \cup \overset{\text{dp}}{\rightarrow})$$

P_0	P_1
(a) `x ← 1`	(b) `r1 ← x`
	(c) `x ← 2`

(a) uniproc Forbidden: `x=1`; `r1=1`;

P_0	P_1
(a) `r1 ← x`	(c) `r4 ← y`
`r9 ← xor r1,r1`	`r9 ← xor r4,r4`
(b) `y ← 1+r9`	(d) `x ← 1+r9`

(b) thin Forbidden: `r1=1`; `r4=1`;



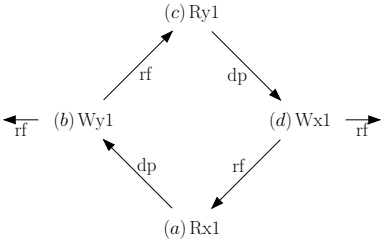


Fig. 3. Invalid executions according to the uniproc and thin criteria

We define the validity of an execution $w.r.t.$ an architecture A as the conjunction of three checks independent of the architecture, namely $\mathrm{wf}(X)$, $\mathrm{uniproc}(X)$ and $\mathrm{thin}(X)$ with a last one that characterises the architecture:

$$A.\mathrm{valid}(X) \triangleq \mathrm{wf}(X) \wedge \mathrm{uniproc}(X) \wedge \mathrm{thin}(X) \wedge \mathrm{acyclic}(A.\mathrm{ghb}(X))$$

2.4 Comparing Architectures *via* Validity Predicates

From our definition of validity arises a simple notion of comparison among architectures. $A_1 \leq A_2$ means that A_1 is *weaker* than A_2:

$$A_1 \leq A_2 \triangleq (\overset{\mathrm{ppo}_1}{\rightarrow} \subseteq \overset{\mathrm{ppo}_2}{\rightarrow}) \wedge (\overset{\mathrm{grf}_1}{\rightarrow} \subseteq \overset{\mathrm{grf}_2}{\rightarrow})$$

The validity of an execution is decreasing $w.r.t.$ the strength of the predicate; *i.e.* a weak architecture exhibits at least all the behaviours of a stronger one:

$$\forall A_1 A_2,\ (A_1 \leq A_2) \Rightarrow (\forall X, A_2^{\epsilon}.\mathrm{valid}(X) \Rightarrow A_1^{\epsilon}.\mathrm{valid}(X))$$

Programs running on an architecture A_1^{ϵ} exhibit executions that would be valid on a stronger architecture A_2^{ϵ}; we characterise all such executions as follows:

$$A_1.\mathrm{check}_{\mathrm{A}_2}(X) \triangleq \mathrm{acyclic}(\overset{\mathrm{grf}_2}{\rightarrow} \cup \overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{ppo}_2}{\rightarrow})$$

These two theorems, though fairly simple, will be useful to compare two models and to restore a strong model from a weaker one, as in Sec. 3.

2.5 Examples

We propose here alternative formulations of *SC* [17] and Sparc's *TSO* [24] in our framework, which we proved equivalent to the original definitions. We omit

proofs and the formal details for lack of space, but they can be found online[2]. We write $\mathrm{po}(X)$ (resp. $\mathrm{rf}(X)$, $\mathrm{rfe}(X)$) for the function extracting the $\overset{\mathrm{po}}{\rightarrow}$ (resp. $\overset{\mathrm{rf}}{\rightarrow}$, $\overset{\mathrm{rfe}}{\rightarrow}$) relation from X. We define notations to extract pairs of memory events from the program order: $MM \triangleq \lambda X.\,((\mathbb{M}\times\mathbb{M})\cap\mathrm{po}(X))$, $RM \triangleq \lambda X.\,((\mathbb{R}\times\mathbb{M})\cap\mathrm{po}(X))$ and $WW \triangleq \lambda X.\,((\mathbb{W}\times\mathbb{W})\cap\mathrm{po}(X))$.

SC allows no reordering of events ($\overset{\mathrm{ppo}}{\rightarrow}$ equals $\overset{\mathrm{po}}{\rightarrow}$ on memory events) and makes writes available to all processors as soon as they are issued ($\overset{\mathrm{rf}}{\rightarrow}$ are global). Thus, there is no need for barriers, and any architecture is weaker than SC: $SC \triangleq (MM, \mathrm{rf}, \lambda X.\emptyset)$. The following criterion characterises, as in Sec. 2.4, valid SC executions on any architecture: $A.\mathrm{check}_{SC}(X) = \mathrm{acyclic}(\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\mathrm{po}}{\rightarrow})$. Thus, the outcome of Fig. 1 will never be the result of an SC execution, as it exhibits the cycle: $(a) \overset{\mathrm{po}}{\rightarrow} (b) \overset{\mathrm{fr}}{\rightarrow} (c) \overset{\mathrm{po}}{\rightarrow} (d) \overset{\mathrm{fr}}{\rightarrow} (a)$.

TSO allows two relaxations [3]: *write to read program order*, meaning its $\overset{\mathrm{ppo}}{\rightarrow}$ includes all pairs but the store-load ones ($ppo_{tso} \triangleq (\lambda X.\,(RM(X) \cup WW(X)))$) and *read own write early* ($\overset{\mathrm{rfi}}{\rightarrow}$ are not global). We elide barrier semantics, detailed in Sec. 3: $TSO^{\epsilon} \triangleq (ppo_{tso}, \mathrm{rfe}, \lambda X.\emptyset)$. Sec. 2.4 shows the following criterion characterises valid executions ($w.r.t.$ any $A \leq TSO$) that would be valid on TSO^{ϵ}, *e.g.* in Fig. 1: $A.\mathrm{check}_{TSO}(X) = \mathrm{acyclic}(\overset{\mathrm{ws}}{\rightarrow} \cup \overset{\mathrm{fr}}{\rightarrow} \cup \overset{\mathrm{rfe}}{\rightarrow} \cup \overset{\text{ppo-tso}}{\rightarrow})$.

3 Semantics of Barriers

We define the semantics and placement in the code that barriers should have to restore a stronger model from a weaker one. It is clearly enough to have $w \overset{\mathrm{ab}_1}{\rightarrow} r$ whenever $w \overset{\mathrm{grf}_2\setminus_1}{\rightarrow} r$ holds to restore store atomicity, *i.e.* a barrier ensuring $\overset{\mathrm{rf}}{\rightarrow}$ is global. But then a processor holding such a barrier placed after r would wait until w is globally performed, then read again to ensure r is globally performed after w. We provide a less costly requirement: when $w \overset{\mathrm{rf}}{\rightarrow} r \overset{\mathrm{po}}{\rightarrow} m$, where r may not be globally performed after w is, inserting a barrier instruction between the instructions generating r and m only forces the processor generating r and m to delay m until w is globally performed.

Formally, given $A_1 \leq A_2$, we define the predicate fb (*fully barriered*) on executions X by

$$A_1.\mathrm{fb}_{A_2}(X) \triangleq \left(\left(\overset{\mathrm{ppo}_2\setminus_1}{\rightarrow}\right) \cup \left(\overset{\mathrm{grf}_2\setminus_1}{\rightarrow}; \overset{\mathrm{ppo}_2}{\rightarrow}\right)\right) \subseteq \overset{\mathrm{ab}_1}{\rightarrow}$$

where $\overset{\mathrm{r}_2\setminus_1}{\rightarrow} \triangleq \overset{\mathrm{r}_2}{\rightarrow} \setminus \overset{\mathrm{r}_1}{\rightarrow}$ is the set difference, and $x \overset{\mathrm{r}_1}{\rightarrow}; \overset{\mathrm{r}_2}{\rightarrow} y \triangleq \exists z.\, x \overset{\mathrm{r}_1}{\rightarrow} z \wedge z \overset{\mathrm{r}_2}{\rightarrow} y$.

The fb predicate provides an insight on the strength that the barriers of the architecture A_1 should have to restore the stronger A_2. They should:

1. restore the pairs that are preserved in the program order on A_2 and not on A_1, which is a static property;

2. compensate for the fact that some writes may not be globally performed at once on A_1 while they are on A_2, which we model by (some subrelation of) $\overset{\text{rf}}{\rightarrow}$ not being global on A_1 while it is on A_2; this is a dynamic property.

We can then prove that the above condition on $\overset{\text{ab}_1}{\rightarrow}$ is sufficient to regain A_2^{ϵ} from A_1:

Theorem 1 (Barrier guarantee)

$$\forall A_1 A_2, (A_1 \leq A_2) \Rightarrow (\forall X, A_1.\text{valid}(X) \wedge A_1.\text{fb}_{A_2}(X) \Rightarrow A_2^{\epsilon}.\text{valid}(X))$$

The static property of barriers is expressed by the condition $\overset{\text{ppo}_2 \setminus 1}{\rightarrow} \subseteq \overset{\text{ab}_1}{\rightarrow}$. A barrier provided by A_1 should ensure that the events generated by a same processor are globally performed in program order if they are on A_2. In this case, it is enough to insert a barrier between the instructions that generate these events.

The dynamic property of barriers is expressed by the condition $\overset{\text{grf}_2 \setminus 1}{\rightarrow}; \overset{\text{ppo}_2}{\rightarrow} \subseteq \overset{\text{ab}_1}{\rightarrow}$. A barrier provided by A_1 should ensure store atomicity to the write events that have this property on A_2. This is how we interpret the *cumulativity* of barriers as stated by Power [21]: the *A-cumulativity* (resp. *B-cumulativity*) property applies to barriers that enforce ordering of pairs in $\overset{\text{rf}}{\rightarrow}; \overset{\text{po}}{\rightarrow}$ (resp. $\overset{\text{po}}{\rightarrow}; \overset{\text{rf}}{\rightarrow}$). We consider a barrier that only preserves pairs in $\overset{\text{po}}{\rightarrow}$ to be *non-cumulative*. Thm. 1 states that, to restore A_2 from A_1, it suffices to insert an A-cumulative barrier between each pair of instructions such that the first one in the program order reads from a write which is to be globally performed on A_2 but is not on A_1.

Restoring SC. We model an A-cumulative barrier as a function returning an ordering relation when given a placement of the barriers in the code:

$$m_1 \overset{\text{fenced}}{\rightarrow} m_2 \quad \triangleq \quad \exists b.\, m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2$$

$$\text{A} - \text{cumul}(X, \overset{\text{fenced}}{\rightarrow}) \quad \triangleq \quad \overset{\text{fenced}}{\rightarrow} \cup \overset{\text{rf}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow}$$

Thm. 1 shows that inserting such a barrier between all $\overset{\text{po}}{\rightarrow}$ pairs restores *SC*:

Corollary 1 (Barriers restoring *SC*)

$$\forall A\ X, (A.\text{valid}(X) \wedge \text{A} - \text{cumul}(X, MM) \subseteq \overset{ab}{\rightarrow}) \Rightarrow SC.\text{valid}(X)$$

Consider *e.g.* the **iriw** test depicted in Fig. 4. The specified outcome may be the result of a non-*SC* execution on a weak architecture in the absence of barriers. Our A-cumulative barrier forbids this outcome, as shown in Fig. 4: if placed between each pair of reads on P_0 and P_1, not only does it prevent their reordering, but also ensures that the write (e) on P_2 (resp. (y) P_3) is globally performed before the second read (b) on P_0 (resp. (d) on P_1).

Thus, we force a program to have an *SC* behaviour by fencing all pairs in $\overset{\text{po}}{\rightarrow}$. Yet, it would be enough to invalidate non-*SC* executions, by fencing only the $\overset{\text{po}}{\rightarrow}$ pairs in the $\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ cycles of these executions. We believe the static analysis of [23] (based on compile-time approximation of $\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ cycles) applies to architectures relaxing store atomicity, if their barriers offer A-cumulativity.

iriw

P_0	P_1	P_2	P_3
(a) r1 ← x	(c) r2 ← y	(e) x ← 1	(f) y ← 2
fence	fence		
(b) r2 ← y	(d) r1 ← x		

Observed? 0:r1=1; 0:r2=0; 1:r2=2; 1:r1=0;

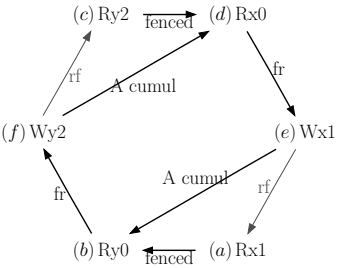


Fig. 4. Study of **iriw** with A-cumulative barriers

4 diy: A Testing Tool

We present our diy (*do it yourself*) tool, which computes litmus tests in x86 or Power assembly code by generating violations of *SC*, *i.e.* cycles in $\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$. A *diy* tutorial is available[1].

4.1 Cycles as Specifications of Litmus Tests

Consider *e.g.* the outcome of Fig. 4 (a): it leads to the $\overset{\text{hb-seq}}{\rightarrow} \cup \overset{\text{po}}{\rightarrow}$ cycle of Fig. 4 (b): from $r_1 = 1$ on P_0, we know the load (a) reads from the store (e) on P_2, thus $(e) \overset{\text{rfe}}{\rightarrow} (a)$. By the `fence` on P_0, we know $(a) \overset{\text{fenced}}{\rightarrow} (b)$ and since $r_2 = 0$ on P_0, we know the load (b) read from the initial state, thus $(b) \overset{\text{fre}}{\rightarrow} (f)$; *idem* on P_1.

The interesting behaviour of a litmus test can be characterised by a cycle formed of relations: *e.g.* the **iriw** test of Fig. 4 can be built from the cycle $\overset{\text{rfe}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow}; \overset{\text{fre}}{\rightarrow}; \overset{\text{rfe}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow}; \overset{\text{fre}}{\rightarrow}$. The computed outcome ensures the input cycle appears in at least one of the execution witnesses of the test. If the outcome is observed, then at least one subsequence in the cycle is not global, *i.e.* not in $\overset{\text{ghb}}{\rightarrow}$: *e.g.* if the `fence` of Fig. 4 orders pairs of loads and since $\overset{\text{ab}}{\rightarrow}$ and $\overset{\text{fr}}{\rightarrow}$ are global, then $\overset{\text{rfe}}{\rightarrow}; \overset{\text{fenced}}{\rightarrow} \not\subseteq \overset{\text{ghb}}{\rightarrow}$, *i.e.* the `fence` is not A-cumulative.

We call sequences of relations *relaxations* and give them a concrete syntax (see Fig. 7 and 8). Thus Rfe represents a $\overset{\text{rfe}}{\rightarrow}$ arrow, Fre a $\overset{\text{fre}}{\rightarrow}$ arrow, and DpdR a $\overset{\text{dp}}{\rightarrow}$ (Dp) arrow targeting a read (R), with different (d) source and target locations.

diy needs to be specified which relaxations are considered global and which are not. When specified a pool of global relaxations, a single non-global relaxation, and a size n (*i.e.* the number of relaxations arrows in the cycle, *e.g.* 6 for **iriw**), diy generates cycles up to size n that contains at least one occurrence of the non-global relaxation. If no non-global relaxation is specified, diy generates cycles up to size n containing the specified global relaxations. When the cycles generation is done, diy computes litmus tests from these cycles, as detailed in the following.

4.2 Code Generation

We show here how we generate a Power litmus test from a given cycle of relaxations by an example below. The complete algorithm for code generation is available online[2]. We write _ for the information not yet set by diy: __ is an undetermined event, $W\_\_$ a write with yet unset location and value, and $Rx\_$ a read from x with undetermined value.

1. Consider *e.g.* the input cycle, issued by diy's cycles generation phase:

$$(a)\_\_ \xrightarrow{\text{Rfe}} (b)\_\_ \xrightarrow{\text{DpdR}} (c)\_\_ \xrightarrow{\text{Fre}} (d)\_\_ \xrightarrow{\text{Rfe}} (e)\_\_ \xrightarrow{\text{DpdR}} (f)\_\_ \xrightarrow{\text{Fre}} (a)$$

.

2. A linear scan sets the directions from the edges. Observe *e.g.* the last edge; $\xrightarrow{\text{Fre}}$ requires a R source and a W target:

$$(a)W\_ \xrightarrow{\text{Rfe}} (b)R\_ \xrightarrow{\text{DpdR}} (c)R\_ \xrightarrow{\text{Fre}} (d)W\_ \xrightarrow{\text{Rfe}} (e)R\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)$$

3. We pick an event e which is the target of a relaxation specifying a location change. If there is none, generation fails. Otherwise, a linear scan starting from e sets the locations. At the end of the scan, if e and its predecessor have the same location (*e.g.* $\xrightarrow{\text{Rfe}} e \xrightarrow{\text{PodRW}}$), generation fails. As $\xrightarrow{\text{DpdR}}$ specifies a location change (*i.e.* we pick (c)), we rewrite the cycle as:

$$(c)R\_ \xrightarrow{\text{Fre}} (d)W\_\_ \xrightarrow{\text{Rfe}} (e)R\_\_ \xrightarrow{\text{DpdR}} (f)R\_\_ \xrightarrow{\text{Fre}} (a)W\_ \xrightarrow{\text{Rfe}} (b)R\_ \xrightarrow{\text{DpdR}} (c)$$

We set the locations starting from (c), changing location between (e) and (f):

$$(c)Rx\_ \xrightarrow{\text{Fre}} ()dWx\_ \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy\_ \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$

4. We cut the input cycle into maximal sequences of events with the same location (*i.e.* $(c)(d)(e)$ and $(f)(a)(b)$), each being scanned *w.r.t.* the cycle order: the first write in each sequence is given value 1, the second one 2, *etc.* The values then reflect the write serialisation order for the specified location:

$$(c)Rx\_ \xrightarrow{\text{Fre}} (d)Wx1 \xrightarrow{\text{Rfe}} (e)Rx\_ \xrightarrow{\text{DpdR}} (f)Ry\_ \xrightarrow{\text{Fre}} (a)Wy1 \xrightarrow{\text{Rfe}} (b)Ry\_ \xrightarrow{\text{DpdR}} (c)$$

5. *Significant reads* are the sources of $\xrightarrow{\text{fr}}$ and the targets of $\xrightarrow{\text{rf}}$ edges. We associate them with the write on the other side of the edge. In the $\xrightarrow{\text{rf}}$ case, the value of the read is the one of its associated write. In the $\xrightarrow{\text{fr}}$ case, the value of the read is the value of the predecessor of its associated write in $\xrightarrow{\text{ws}}$, *i.e.* by construction the value of its associated write minus 1. Non significant reads do not appear in the test condition. All the reads are significant here:

$$(c)Rx0 \xrightarrow{\text{Fre}} (d)Wx1 \xrightarrow{\text{Rfe}} (e)Rx1 \xrightarrow{\text{DpdR}} (f)Ry0 \xrightarrow{\text{Fre}} (a)Wy1 \xrightarrow{\text{Rfe}} (b)Ry1 \xrightarrow{\text{DpdR}} (c)$$

6. We generate the litmus test given in Fig. 5 for Power.

```
{ 0:r2=y; 0:r5=x; 1:r2=x; 2:r2=x; 2:r5=y; 3:r2=y; }

 P0               | P1               | P2               | P3
(b) lwz r1,0(r2)  |     li r1,1      | (e) lwz r1,0(r2) |    li r1,1
    xor r3,r1,r1  | (d) stw r1,0(r2) |     xor r3,r1,r1 |(a) stw r1,0(r2)
(c) lwzx r4,r3,r5 |                  | (f) lwzx r4,r3,r5 |

exists (0:r1=1 /\ 0:r4=0 /\ 2:r1=1 /\ 2:r4=0)
```

Fig. 5. iriw with dependencies in Power assembly

We add *e.g.* a `xor` instruction between the instructions associated with the events (b) and (c) to implement the dependency required by the $\overset{\text{DpdR}}{\longrightarrow}$ relation between them.

The test in Fig. 5 actually is a Power implementation of **iriw** [9] with dependencies. `diy` recovers indeed many classical tests, such as **rwc** [9] (see also Fig. 8).

5 Case Study: The Power Architecture

We now instantiate the formalism of Sec. 2 for Power by adding *register events* to reflect register accesses [22], and *commit events* to express branching decisions. $\mathbb{C}$ is the set of commits, and c is an element of $\mathbb{C}$. We handle three barrier instructions : `isync`, `sync` and `lwsync`. We distinguish the corresponding events by the eponymous predicates, *e.g.* is-isync. An execution witness includes an additional *intra-instruction causality* relation $\overset{\text{iico}}{\rightarrow}$: *e.g.* executing the indirect load `lwz r1, 0(r2)` (`r2` holding the address of a memory location `x` containing 1) creates three events $(a)\,\mathrm{R}r_2\mathrm{x}$, $(b)\,\mathrm{Rx1}$ and $(c)\,\mathrm{W}r_1 1$, such that $(a) \overset{\text{iico}}{\rightarrow} (b) \overset{\text{iico}}{\rightarrow} (c)$. Moreover, $\overset{\text{rf}}{\rightarrow}$ now also relates register events: we write $\overset{\text{rf-reg}}{\rightarrow}$ the subrelation of $\overset{\text{rf}}{\rightarrow}$ relating register stores to register loads that read their values.

Preserved program order. We present in Fig. 6(a) the definition of $\overset{\text{ppo-ppc}}{\rightarrow}$, induced by lifting the ordering constraints of a processor to the global level (where $+$ is the transitive closure). This is a formal presentation of the *data dependencies* ($\overset{\text{dd}}{\rightarrow}$) and *control dependencies* ($\overset{\text{ctrl}}{\rightarrow}$ and $\overset{\text{isync}}{\rightarrow}$) of [21, p. 661] which allows loads to be speculated if no `isync` is added after the branch but prevents stores from being speculated in any case. This is similar to Sparc RMO [24, V9, p. 265].

Read-from maps. Since Power allows store buffering [21, p.661], $\overset{\text{rfi}}{\rightarrow}$ is not global. Running **iriw** with data dependencies (Fig. 5) on Power reveals that $\overset{\text{rfe}}{\rightarrow}$ is not global either. This is the main particularity of the Power architecture.

Barriers. We define in Fig. 6 (b) the `sync` barrier [21, p. 700] as the *SC*-restoring A-cumulative barrier of Sec. 3 extended to B-cumulativity. Power features another cumulative barrier [21, p. 700], `lwsync`, defined in Fig. 6 (b). `lwsync` acts as `sync` except on store-load pairs, in both the base and cumulativity cases.

$$\overset{\text{dd}}{\rightarrow} \triangleq (\overset{\text{rf-reg}}{\rightarrow} \cup \overset{\text{iico}}{\rightarrow})^+ \qquad r \overset{\text{ctrl}}{\rightarrow} w \triangleq \exists c \in \mathbb{C}.\ r \overset{\text{dd}}{\rightarrow} c \overset{\text{po}}{\rightarrow} w$$

$$r \overset{\text{isync}}{\rightarrow} e \triangleq \exists c \in \mathbb{C}.\ r \overset{\text{dd}}{\rightarrow} c \wedge \exists b.\ \text{is-isync}(b) \wedge c \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} e$$

$$\overset{\text{dp}}{\rightarrow} \triangleq \overset{\text{ctrl}}{\rightarrow} \cup \overset{\text{isync}}{\rightarrow} \cup \left(\left(\overset{\text{dd}}{\rightarrow} \cup (\overset{\text{po-loc}}{\rightarrow} \cap (\mathbb{W} \times \mathbb{R})) \right)^+ \cap (\mathbb{R} \times \mathbb{M}) \right) \qquad \overset{\text{ppo-ppc}}{\rightarrow} \triangleq \overset{\text{dp}}{\rightarrow}$$

(a) Preserved program order

$$m_1 \overset{\text{sync}}{\rightarrow} m_2 \triangleq \exists b.\ \text{is-sync}(b) \wedge m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2$$

$$m_1 \overset{\text{ab-sync}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{sync}}{\rightarrow} m_2 \vee \exists r.\ m_1 \overset{\text{rf}}{\rightarrow} r \overset{\text{ab-sync}}{\rightarrow} m_2 \vee \exists w.\ m_1 \overset{\text{ab-sync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} m_2$$

(b) Barrier sync

$$m_1 \overset{\text{lwsync}}{\rightarrow} m_2 \triangleq \exists b.\ \text{is-lwsync}(b) \wedge m_1 \overset{\text{po}}{\rightarrow} b \overset{\text{po}}{\rightarrow} m_2$$

$$m_1 \overset{\text{ab-lwsync}}{\rightarrow} m_2 \triangleq m_1 \overset{\text{lwsync}}{\rightarrow} m_2 \cap ((\mathbb{W} \times \mathbb{W}) \cup (\mathbb{R} \times \mathbb{M})) \vee \exists r.\ m_1 \overset{\text{rf}}{\rightarrow} r \overset{\text{ab-lwsync}}{\rightarrow} m_2 \wedge m_2 \in \mathbb{W} \vee \exists w.\ m_1 \overset{\text{ab-lwsync}}{\rightarrow} w \overset{\text{rf}}{\rightarrow} m_2 \wedge m_1 \in \mathbb{R}$$

(c) Barrier lwsync

$$\overset{\text{ab-ppc}}{\rightarrow} \triangleq \overset{\text{ab-sync}}{\rightarrow} \cup \overset{\text{ab-lwsync}}{\rightarrow}$$

$$\mathit{Power} \triangleq (\overset{\text{ppo-ppc}}{\rightarrow}, \emptyset, \overset{\text{ab-ppc}}{\rightarrow})$$

Fig. 6. A Power model

Experiment. diy generated 800 Power tests and ran them up to 1e12 times each on 3 machines: `squale`, a 4-processor Power G5 running Mac OS X, `hpcx` a Power 5 with 16 processors per node and `vargas`, a Power 6 with 32 processors per node, both of them running AIX. The detailed protocol and results are available[1].

Following our model, we assumed $\overset{\text{ws}}{\rightarrow}$, $\overset{\text{fr}}{\rightarrow}$, $\overset{\text{ppo-ppc}}{\rightarrow}$ and $\overset{\text{ab-ppc}}{\rightarrow}$ to be global and tested it by computing *safe* tests whose input cycles only include relaxations we suppose global, *e.g.* $\overset{\text{SyncdWW}}{\longrightarrow}; \overset{\text{Wse}}{\longrightarrow}; \overset{\text{SyncdWR}}{\longrightarrow}; \overset{\text{Fre}}{\longrightarrow}$. We ran the tests supposed to, according to our model, exhibit relaxations. These tests are given in Fig. 7 (where M stands for million). We observed all of them at least on one machine, which corresponds with our model. For each relaxation observed on a given machine, we write the highest number of outcomes. When a relaxation is not observed, we write the total of outcomes: thus we write *e.g.* 0/16725M for PodRR on `vargas`.

For each machine, we observed the number of runs required to exhibit the least frequent relaxation (*e.g.* 32 million for Rfe on `vargas`), and ran the safe tests at least 20 times this number. The outcomes of the safe tests have not been observed on `vargas` and `squale`, which increases our confidence in the safe set we assumed. Yet, `hpcx` exhibits non-*SC* behaviours for some A-cumulativity tests, including classical ones [9] like **iriw** with `sync` instructions on P_0 and P_1 (see Fig. 8). We understand that this is due to an erratum in the Power 5 implementation. IBM is providing a workaround, replacing the `sync` barrier by a short code sequence [Personal Communication], and our testing suggests this does regain *SC* behaviour for the examples in question (e.g. with 0/4e10 non-*SC* results for **iriw**). We understand also that the erratum should not be observable

Relaxation	Definition[a]	hpcx	squale	vargas
PosRR	$r_\ell \xrightarrow{\text{po}} r'_\ell$	2/40M	3/2M	0/4745M
PodRR	$r_\ell \xrightarrow{\text{po}} r'_{\ell'}$	2275/320M	12/2M	0/16725M
PodRW	$r_\ell \xrightarrow{\text{po}} w'_{\ell'}$	8653/400M	178/2M	0/6305M
PodWW	$w_\ell \xrightarrow{\text{po}} w'_{\ell'}$	2029/4M	2299/2M	2092501/32M
PodWR	$w_\ell \xrightarrow{\text{po}} r'_{\ell'}$	51085/40M	178286/2M	672001/32M
Rfi	$\xrightarrow{\text{rfi}}$	7286/4M	1133/2M	145/32M
Rfe	$\xrightarrow{\text{rfe}}$	177/400M	0/1776M	9/32M
LwSyncsWR	$w_\ell \xrightarrow{\text{lwsync}} r'_\ell$	243423/600M	2/40M	385/32M
LwSyncdWR	$w_\ell \xrightarrow{\text{lwsync}} r'_{\ell'}$	103814/640M	11/2M	117670/32M
ACLwSyncsRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_\ell$	11/320M	0/960M	1/21M
ACLwSyncdRR	$w_\ell \xrightarrow{\text{rfe}} r'_\ell \xrightarrow{\text{lwsync}} r''_{\ell'}$	124/400M	0/7665M	2/21M
BCLwSyncsWW	$w_\ell \xrightarrow{\text{lwsync}} w'_\ell \xrightarrow{\text{rfe}} r''_\ell$	68/400M	0/560M	2/160M
BCLwSyncdWW	$w_\ell \xrightarrow{\text{lwsync}} w'_{\ell'} \xrightarrow{\text{rfe}} r''_{\ell'}$	158/400M	0/11715M	1/21M

[a] Notation: r_ℓ (w_ℓ) is a read (write) event with location ℓ.

Fig. 7. Selected results of the diy experiment matching our model

Cycle	hpcx	Name in [9]
Rfe SyncdRR Fre Rfe SyncdRR Fre	2/320M	iriw
Rfe SyncdRR Fre SyncdWR Fre	3/400M	rwc
DpdR Fre Rfe SyncsRR DpdR Fre Rfe SyncsRR	1/320M	
Wse LwSyncdWW Wse Rfe SyncdRW	1/800M	
Wse SyncdWR Fre Rfe LwSyncdRW	1/400M	

Fig. 8. Anomalies observed on Power 5

for conventional lock-based code and that Power 6 is not subject to it; the latter is consistent with our testing on vargas.

6 Related Work

Formal memory models roughly fall into two classes: operational models and axiomatic models. Operational models, e.g. [25,15], are abstractions of actual machines composed of idealised hardware components such as queues. They can be appealingly intuitive and offer a relatively direct path to simulation, at least in principle. Axiomatic models focus on segregating allowed and forbidden behaviours, usually by constraining various order relations on memory accesses; they are particularly well adapted for model exploration, as we do here. Several of the more formal vendor specifications have been in this style [5,24,16].

One generic axiomatic model related to ours is Nemos [26]. This covers a broad range of models including Itanium as the most substantial example. Itanium is rather different to Power; we do not know whether our framework could handle

such a model or whether a satisfactory Power model could be expressed in Nemos. By contrast, our framework owes much to the concept of *relaxation*, informally presented in [3]. As regards tools, Nemos calculates the behaviour of example programs w.r.t. to a model, but offers no support for generating or running tests on actual hardware.

Previous work on model-building based on experimental testing includes that of Collier [12] and Adir et al. [2,1]. The former is based on hand-coded test programs and Collier's model, in which the cumulativity of the Power barriers does not seem to fit naturally. The latter developed an axiomatic model for a version of Power before cumulative barriers [1]; their testing [2] aims to produce interesting collisions (accesses to related locations) with knowledge of the microarchitecture, using an architecture model as an oracle to determine the legal results of tests rather than (as we do) generating interesting tests from the memory model.

7 Conclusion

We present here a general class of axiomatic memory models, extending smoothly from *SC* to a highly relaxed model for Power processors. We model their relaxation of store atomicity without requiring multiple write events per store [16], or a view order per processor [12,1,21,6]. Our principal validity condition is simple, just an acyclicity check of the global happens before relation. This check is already known for *SC* [18], and recent verification tools use it for architectures with store buffer relaxation [14,10]. Our Power model captures key aspects of the behaviour of cumulative barriers, though we do not regard it as definitive: on the one hand there are known tests for which the model is too weak w.r.t. our perception of the architect's intent (particularly involving the lightweight barrier `lwsync`); on the other hand, given that we rely heavily on black-box testing, it is hard to establish confidence that there are not tests that would invalidate our model. Despite that, our automatic test generation based on the model succeeds in generating interesting tests, revealing a rare Power 5 implementation erratum for barriers in lock-free code. This is a significant advance over reliance on hand-crafted litmus tests.

Acknowledgements. We thank Damien Doligez and Xavier Leroy for invaluable discussions and comments, Assia Mahboubi and Vincent Siles for advice on the Coq development, and Thomas Braibant, Jules Villard and Boris Yakobowski for comments on a draft. We thank the HPCx (UK) and IDRIS(.fr) high-performance computing services. We acknowledge support from EPSRC grants EP/F036345 and EP/H005633 and ANR grant ANR-06-SETI-010-02.

References

1. Adir, A., Attiya, H., Shurek, G.: Information-Flow Models for Shared Memory with an Application to the PowerPC Architecture. In: TPDS (2003)
2. Adir, A., Shurek, G.: Generating Concurrent Test-Programs with Collisions for Multi-Processor Verification. In: HLDVT (2002)

3. Adve, S.V., Gharachorloo, K.: Shared Memory Consistency Models: A Tutorial. IEEE Computer 29, 66–76 (1995)
4. Ahamad, M., Bazzi, R.A., John, R., Kohli, P., Neiger, G.: The Power of Processor Consistency. In: SPAA (1993)
5. Alpha Architecture Reference Manual, 4th edn. (2002)
6. ARM Architecture Reference Manual (ARMv7-A and ARMv7-R) (April 2008)
7. Arvind, Maessen, J.-W.: Memory Model = Instruction Reordering + Store Atomicity. In: ISCA, IEEE Computer Society, Los Alamitos (2006)
8. Bertot, Y., Casteran, P.: Coq'Art. EATCS Texts in Theoretical Computer Science. Springer, Heidelberg (2004)
9. Boehm, H.-J., Adve, S.V.: Foundations of the C++ Concurrency Memory Model. In: PLDI (2008)
10. Burckhardt, S., Musuvathi, M.: Effective Program Verification for Relaxed Memory Models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
11. Cantin, J., Lipasti, M., Smith, J.: The Complexity of Verifying Memory Coherence. In: SPAA (2003)
12. Collier, W.W.: Reasoning About Parallel Architectures. Prentice-Hall, Englewood Cliffs (1992)
13. Dubois, M., Scheurich, C.: Memory Access Dependencies in Shared-Memory Multiprocessors. IEEE Transactions on Software Engineering 16(6) (June 1990)
14. Hangal, S., Vahia, D., Manovit, C., Lu, J.-Y.J., Narayanan, S.: TSOTool: A Program for Verifying Memory Systems Using the Memory Consistency Model. In: ISCA (2004)
15. Higham, L., Kawash, J., Verwaal, N.: Weak memory consistency models part I: Definitions and comparisons. Technical Report98/612/03, Department of Computer Science, The University of Calgary (January 1998)
16. A Formal Specification of Intel Itanium Processor Family Memory Ordering . Intel Document 251429-001 (October 2002)
17. Lamport, L.: How to Make a Correct Multiprocess Program Execute Correctly on a Multiprocessor. IEEE Trans. Comput. 46(7), 779–782 (1979)
18. Landin, A., Hagersten, E., Haridi, S.: Race-free interconnection networks and multiprocessor consistency. SIGARCH Comput. Archit. News 19(3), 106–115 (1991)
19. Manson, J., Pugh, W., Adve, S.V.: The Java Memory Model. In: POPL (2005)
20. Owens, S., Sarkar, S., Sewell, P.: A Better x86 Memory Model: x86-TSO. In: TPHOL (2009)
21. Power isa version 2.06 (2009)
22. Sarkar, S., Sewell, P., Zappa Nardelli, F., Owens, S., Ridge, T., Braibant, T., Myreen, M., Alglave, J.: The Semantics of x86-CC Multiprocessor Machine Code. In: POPL (2009)
23. Shasha, D., Snir, M.: Efficient and Correct Execution of Parallel Programs that Share Memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
24. Sparc Architecture Manual Versions 8 and 9 (1992/1994)
25. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an Operational Memory Model Specification Framework with Integrated Model Checking Capability. In: CCPE (2007)
26. Yang, Y., Gopalakrishnan, G., Linstrom, G., Slind, K.: Nemos: A Framework for Axiomatic and Executable Specifications of Memory Consistency Models. In: IPDPS (2004)

Generating Litmus Tests for Contrasting Memory Consistency Models*

Sela Mador-Haim, Rajeev Alur, and Milo M.K. Martin

University of Pennsylvania

Abstract. Well-defined memory consistency models are necessary for writing correct parallel software. Developing and understanding formal specifications of hardware memory models is a challenge due to the subtle differences in allowed reorderings and different specification styles. To facilitate exploration of memory model specifications, we have developed a technique for systematically comparing hardware memory models specified using both operational and axiomatic styles. Given two specifications, our approach generates all possible multi-threaded programs up to a specified bound, and for each such program, checks if one of the models can lead to an observable behavior not possible in the other model. When the models differs, the tool finds a minimal "litmus test" program that demonstrates the difference. A number of optimizations reduce the number of programs that need to be examined. Our prototype implementation has successfully compared both axiomatic and operational specifications of six different hardware memory models. We describe two case studies: (1) development of a non-store atomic variant of an existing memory model, which illustrates the use of the tool while developing a new memory model, and (2) identification of a subtle specification mistake in a recently published axiomatic specification of TSO.

1 Introduction

Well-defined memory consistency models are necessary for writing correct and efficient shared memory programs [1]. The emergence of mainstream multicore processors as well as recent developments in language-level memory models [3,18], have stirred new interest in hardware-level memory models. The formal specification of memory models is challenging due to the many subtle differences between them. Examples of such differences include different allowed reorderings, store atomicity, types of memory fences, load forwarding, control and data dependencies, and different specification styles (operational and axiomatic). Architecture manuals include litmus tests that can be used to differentiate between memory models [15,22], but these litmus tests are not complete, and coming up

* The authors acknowledge the support of NSF grants CCF-0905464 and CCF-0644197, and of the Gigascale Systems Research Center, one of six research centers funded under the Focus Center Research Program (FCRP), a Semiconductor Research Corporation entity.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 273–287, 2010.

with new litmus tests requires identifying the subtle difference between memory models this test is meant to detect.

Our goal is to aid the process of developing specifications for hardware-level memory models by providing a technique for systematically comparing memory model specifications. When there is a difference between the two memory models, the technique generates a litmus test as a counter-example, including both a program and an outcome allowed only in one of the models. Such a technique can be used in several different scenarios. One case is comparing two presumably equivalent models, for example comparing an axiomatic specification given as a set of first order logic formulas to an operational specification that describes the model as a state transition system. Alternatively, we may also want to check whether one model is strictly weaker (or stronger) than the other.

Our approach is based on systematic generation of all possible programs up to a specified size bound. For each program, we check if one of the models can lead to an observable behavior that is not possible in the other model. To produce the set of observable behaviors for a program under a given memory model, we use two different search techniques depending on whether the model specification is operational or axiomatic. When there is an observable behavior in one memory model that is not allowed by the other model, the approach outputs the program and the contrasting behavior. Because we explore starting with the smallest programs, this is a minimal litmus test.

We employ several techniques to make this approach practical. A naive enumeration of all test programs up to the specified bound produces too many programs, so we employ optimizations to reduce the number of programs that need to be examined. We use symmetry reductions based on value, address and thread symmetries. Furthermore, we identify and skip redundant programs that will not expose any new differences by analyzing the conflict graph of the program. We use partial order reduction techniques to optimize exploration of operational models and an incremental SAT approach for axiomatic models.

We tested this approach by comparing the axiomatic and operational specifications of six different memory models: Sequential Consistency (SC), SPARC's TSO, PSO and RMO [22] and non-store-atomic relaxations of TSO and PSO. Our technique finds the known differences, but it also uncovered some errors in two of our specifications, which we corrected. Finding differences takes less than a second in most cases and only several minutes in the worst cases we encountered. We tested the scalability of this technique and found that we can explore all programs up to six read and write operations plus any number of fences in a few minutes. Our results indicate these bounds are adequate to detect subtle differences.

We performed two case studies. We developed a specification of a non-store-atomic variant of PSO, which illustrates that the tool quickly identifies subtle specification mistakes. In another case study, we contrasted SOBER's axiomatic specification of TSO [5] with an operational specification of TSO and showed our technique detects a recently discovered specification error [7].

Initially $X = 0$; $Y = 0$

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 1$
Read $Y \rightarrow r1$	Read $X \rightarrow r2$

Is the outcome $r1 = 0$; $r2 = 0$ allowed?

Fig. 1. Testing write-after-read reordering

2 Specifying Memory Models

A *memory consistency model* is a specification of the shared memory semantics of a parallel system [1]. The simplest memory model is *Sequential Consistency* (SC) [16]. An execution of a concurrent program is sequentially consistent if all reads and writes appear to have occurred in a sequential order that is in agreement with the individual program orders of each thread. In order to improve system performance and allow common hardware optimization techniques such as store buffers, many systems implement *weaker memory models* such as SPARC's TSO, PSO and RMO [22], Intel's x86 [15], Intel's Itanium [24], ARM and PowerPC [2].

Consider for example the program in Fig. 1. Executing under SC, at least one of the writes must occur before any of the reads, and therefore the outcome $r1 = 0$; $r2 = 0$ is not allowed. A processor that has a store buffer, on the other hand, can defer the writes to the main memory and effectively reorder the writes after the reads, and thus reading zero for both registers is allowed. SPARC's TSO and x86 both allow this relaxation. Other memory models allow further relaxations such as write after write and read after read (RMO, Itanium, PowerPC). Some memory models such as SC are *store atomic*, in the sense that all threads observe writes in the same order, but other memory models are non-store-atomic and allow different threads to observe writes from other threads in a different order (such as PowerPC).

2.1 Operational Specification

The purpose of a memory model specification is to express precise constraints on which values can be associated with reads in a given multi-threaded program. One method of specifying a memory model is using an operational style, which abstracts actual hardware structures such as a store buffer. This section describes operational specifications for several memory models that we defined as a part of this work.

For example, we have specified TSO using three different types of components that run concurrently [17]. A processor component produces a sequence of memory operations, a memory location component keeps track of the latest value written to a specific memory location, and a write queue component implements a FIFO buffer for write messages, and supports read forwarding. These components are connected in the configuration described in Fig. 2 (left) to implement

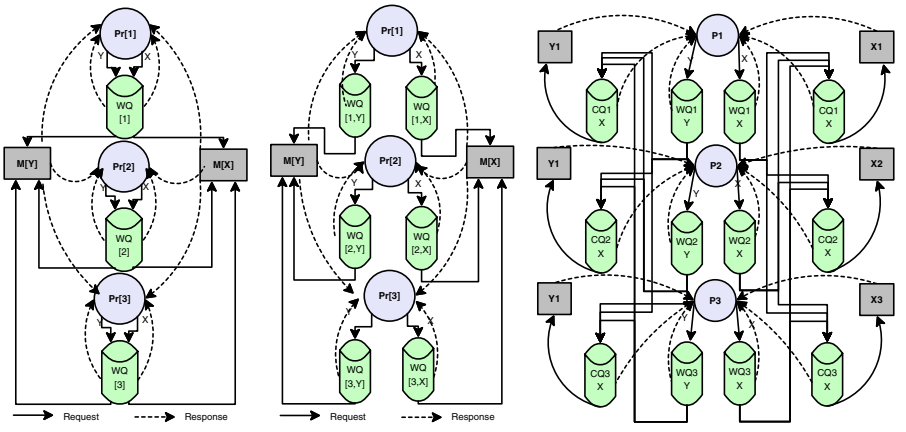


Fig. 2. Component diagram of TSO (left), PSO (middle) and NPSO (right)

TSO. Each processor is connected to a single write queue (WQ) that releases writes from this processor to the main memory.

SPARC's PSO (Partial Store Order [22]), a memory model that relaxes TSO by allowing to reorder writes after writes to different addresses. It can be specified in a similar manner, using the configuration illustrated in Fig. 2 (middle). Instead of one queue per processor, there is a queue per address for each of the processors. Writes to different addresses are stored at different queues, which can send the writes in any order, thus enabling reordering writes after later writes.

The two previous models are store atomic (all threads observe writes from other threads in the same order). In non-store-atomic memory models, different threads do not have to agree on the order of writes from other threads. As an example for a non-store-atomic memory model we present here NPSO, the non-store-atomic version of PSO. The diagram in Fig. 2 (right) presents the operational specification for NPSO. Because each thread may observe stores from different threads in a different order, the NPSO specification does not use one main memory as in the previous models. Instead, each thread has its own local memory. To preserve coherence and ensure all writes to the same address would be observed in a total order. The model maintains coherence by using an additional layer of write queues.

We have defined operational specifications in this style for additional memory models [17] such as RMO and the non-store-atomic versions of each of the store atomic models (NTSO and NPSO). To model these, we add additional component types. The encoding for RMO, for example, requires the ability to read future values to reorder reads after later writes.

2.2 Axiomatic Specification

An alternative approach is the axiomatic style of specifications, given by a set of axioms that define which *execution traces* are allowed by the model and

in particular which writes can be observed by each read. An execution trace is a sequence of memory operations (Read, Write, Fence) produced by a program. Each operation in the trace includes an identifier of the thread that produced this operation, and the address and value of the operation for reads and writes.

Axiomatic specifications usually refer to the program order, $<_p$. For two operations x and y, $x <_p y$ if both x and y belong to the same thread and x precedes y in the execution trace. The program order, however, is not necessarily the order in which memory operations are observed by the main memory. The memory order, $<_m$, is a total order that indicates the order in which memory operations affect the main memory. A read observes the latest write to the same address according to $<_m$.

We define store atomic memory models using two types of axioms: a *read-values* axiom and an *ordering* axiom. The *read-values* axiom states that each read observes the latest write to the same location according to the memory order. To support load forwarding, reads may observe local writes that precede them in program order, even if such write is ordered after the read in the memory order. We handle this forwarding in same style as in Burckhardt et al [4], by defining a function $sees(x, y, <)$, which is true if y is a write and $y < x$ or $y <_p x$. The *read-values* axiom for store-atomic memory models is:

Read values. Given a read x and a write y to the same address as x, then x and y have the same value if $sees(x, y, <_m)$ and there is no other write z such that $sees(x, z, <_m)$ and $y <_m z$. If for a read x there is no write y such that $sees(x, y, <_m)$ then the read value is 0.

All our store atomic memory model specifications use the same read-values axiom, but differ in the definition of the *ordering* axiom, specifying which memory orders are allowed by the model. For example, TSO allows reordering only writes after later reads, and therefore the TSO reordering axiom is:

TSO-reordering. For every x and y, $x <_p y$ implies that $x <_m y$, unless x is a write and y is a read.

The ordering axiom for PSO relaxes TSP by allowing reordering writes with other writes to a different location. The ordering axiom for PSO is:

PSO-reordering. For every x and y, if $x <_p y$ then $x <_m y$ in the following cases: 1. x is a read. 2. Either x or y is a fence. 3. Both x and y are writes and they both have the same address.

In non-store-atomic models, threads may observe stores in different orders, so we can no longer use one global memory order. Instead, we define an order $<_t$ for each thread t, which we call the *view* of thread t. To ensure transitive causal order between operations, the view includes all operations and not only writes.

As in the store-atomic case, loads see the latest stores to the same address except in the case of forwarding, but the relevant order for loads in thread t is

view order $<_t$. We modify the read-values and ordering axioms to observe the latest write in the relevant view:

Non-store-atomic read-values. Given a read x in thread t and a write y to the same address as x, then x and y have the same value if y is the most recent write according to $sees(x, y, <_t)$. If for a read x in thread t there is no write y such that $sees(x, y, <_t)$, the read value is 0.

To define NPSO, the non-store atomic version of PSO maintains the same order restrictions between operation from the same thread as in the case of PSO:

NPSO ordering. For every x and y, if $x <_p y$ then for every t $x <_t y$ must hold in the following cases: 1. x is a read. 2. Either x or y is a fence. 3. Both x and y are writes and they both have the same address.

The non-store-atomic case requires adding another axiom for coherence, stating that there is a total order between writes to the same address:

NPSO coherence. For every two write operations x and y that write to the same address, and for every two threads, t and t', if $x <_t y$ then $x <'_t y$.

The above axioms represent our first attempt at specifying a model which is a non-store atomic relaxation of PSO in an axiomatic style, but, as we describe in Section 4, this specification is too weak. In Section 4.3, we use our technique to develop the missing axioms for NPSO.

3 Comparing Memory Models

This section presents a technique for comparing memory models. Our goal is to check the difference between two models, and when the two models are not equivalent, to generate a litmus test that shows the difference between the two. Two memory models M and M' are not equivalent if any program displays different behaviors under M and M'.

Based on a review of published litmus tests in the literature and our own experience, tests that detect differences between memory models tend to be small, and hence an exhaustive search of test programs up to a given bound is a plausible approach for debugging memory model specifications. Given upper bounds for the total number of instructions in a program, the number of operations per thread, the number of threads as well as the number of memory locations, the technique exhaustively explores all programs within these bounds.

We start by defining the test program space for contrasting memory models. We present reduction techniques for trimming down the number of programs to a manageable size. Finally, we discuss techniques to efficiently compare the set of possible outcomes for a given program both for operational and axiomatic specification styles.

Test A

T1	T2
Write $Y \leftarrow 1$	Read $X \rightarrow r1$
Read $Y \rightarrow r2$	Fence
Write $X \leftarrow 2$	Read $Y \rightarrow r3$

Test B

T1	T2
Write $X \leftarrow 1$	Read $Y \rightarrow r1$
Read $X \rightarrow r2$	Fence
Write $Y \leftarrow 2$	Read $X \rightarrow r3$

Test C

T1	T2
Read $Y \rightarrow r1$	Write $X \leftarrow 1$
Fence	Read $X \rightarrow r2$
Read $X \rightarrow r3$	Write $Y \leftarrow 2$

Fig. 3. Address symmetry (A and B); Thread symmetry (B and C)

3.1 Test Programs

A test program is a concurrent program consisting of n threads, $t_1, ...t_n$, where each thread is a sequence of memory operations. A memory operation can be one of:

- Read $Addr \rightarrow reg$ - a read from a constant address to a register
- Write $Addr \leftarrow Val$ - a write of a constant value to a constant address
- Fence - a full memory ordering barrier (fence)

The above three instructions suffice to contrast the models we have considered in this paper. Our methodology as well as the tool can be extended to include other instructions and data dependencies.

3.2 Program Enumeration

Even when considering small bounds on test size, the program space can be too big to be explored in a reasonable time. Thus, we reduce the number of tested programs to a smaller number of representatives that are still sufficient for finding differences. First, because all writes are constants, registers in the program are used only for defining the final outcome. Therefore, we assign a unique register to each read. Likewise, the actual values read or written are inconsequential. We are interested only in which stores each load instruction can read. So instead of exploring all different combinations of write values, we assign a unique value for each write. We also restrict the places where we add fences: fences at the beginning or end of a thread have no effect, nor does a fence followed by another fence, so we eliminate all fences that are not between two other instructions.

Next, we use the symmetry properties of the memory model to reduce the number of programs. We use two symmetries: address symmetry and thread symmetry. In Fig. 3, the two programs display address symmetry: we obtain Test B from Test A by switching the Xs with the Ys. These two programs display the same behaviors and therefore it is sufficient to test only one of them. Similarly, Test C is the same as Test B with thread T1 switched with T2. By transitivity, any combination of thread and address permutation are equivalent. Hence, Test A and Test C are also symmetric.

We generate only one representative for each symmetry class by assigning an order between elements in a permutation and sorting them, and then we generate programs with sorted elements only. We sort the addresses according to the order of their appearance in the program, starting from T1 and continuing to the next thread after the end of each thread: the first memory access in T1

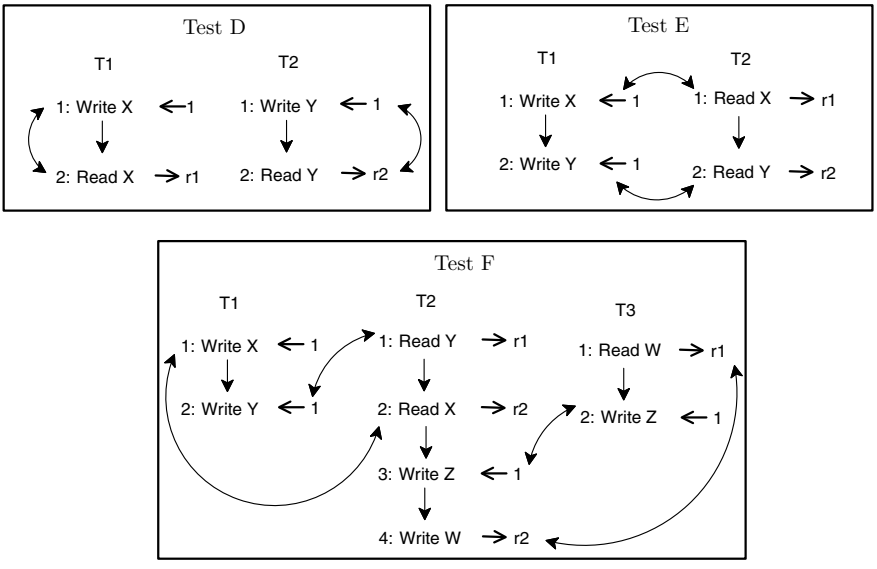


Fig. 4. Redundant tests

is always to location 0, the next memory access could either be to 0 again or to 1 and so on. When the highest address accessed so far is i, the next memory operation involves any address between 0 to $i+1$. Similarly, we perform thread symmetry reduction by sorting threads according to some lexicographical order between instructions. The order we use is $Write < Read < Fence$, where two writes (or reads) are sorted according to their address. By generating programs so that the threads are sorted according to this lexicographical order and addresses by the order of their appearance, the enumeration algorithm avoids generating symmetric tests.

3.3 Redundant Test Elimination

Some test programs are redundant in the sense that these tests are either not going to detect any difference between memory models or are subsumed by smaller programs that detect the same difference. First, we conclude that some programs are redundant simply by looking at the program structure. Consider, for example, Test D in Fig. 4. In this case, there are no shared variables between the two threads, and any execution under any memory model would give the same outcome. Similarly, in Test E both variables are shared, but even SC (the strongest model we typically consider) allows all possible outcomes. In both tests, there is no possible conflict in SC and therefore no cases that could be relaxed under a weaker memory model. Furthermore, consider Test F in Fig 4. This test can be decomposed into two separate tests: Test F1 includes T1 and the first two instructions in T2, and test F2 includes the last two instructions in T2 and T3. Test F is not going to exhibit any behaviors that can not be detected by F1

and F2, because the only relation between the two is the program order relation between instruction 2 and 3 in T2.

We eliminate such redundant test programs by generating a *conflict graph* for the test program. A conflict graph G is a directed graph where each operation is a node and the edges represent potential conflicts between the operations. For every two operations, X and Y, there is an edge in G from X to Y if either: (1) $X <_p Y$, or (2) either of X or Y are write operations and both access the same address. A test is redundant if the conflict graph G for this test is not strongly connected, i.e., there are operations X and Y in the graph such that there is no path from X to Y. For example, in Test C, there is no path from instruction 3 to instruction 2 in T2, and therefore this test is redundant.

Given a program P whose conflict graph in not strongly connected, we partition the instructions in P into two partitions, P_1 and P_2, such that no variables are shared between P_1 and P_2, and if x is an instruction in P_1 and y is an instruction in P_2 and both x and y are in the same thread, then $x <_p y$. We expect that for such a program, no instruction in P_1 would interfere with the execution of P_2 and vice versa, and hence the cross product of the outcomes of the program in partition P_1 and the outcomes of the program in partition P_2 is the set of outcomes of P. Therefore, if P detects a difference between two models, either P_1 or P_2 should detect a difference as well.

3.4 Computing All Outcomes of a Test Program

For each of the test programs we determine if the set of outcomes of P running under a memory model M is the same as for P running on M'. The approach we take is to find all possible outcomes under both models independently and then compare them.

Finding all outcomes for an operational memory model is done in a manner similar to Park and Dill [21]. We use a model checker to find the reachable state space of the model. We extract the outcomes from the set of reachable final states found by the model checker. Our initial experiences in translating the operational models into Promela and running Spin [14] resulted in an inefficient exploration tool. Consequently, we implemented a custom explicit state enumeration model-checker in C++ using sleep-set partial order reduction [12] and state caching.

For memory models specified axiomatically, the model is translated into a propositional formula. The model is specified as a set of first order formulas. In the context of finite programs all the variables have finite domains, so we convert the specification into predicate calculus by unfolding the quantifiers. A satisfying assignment is obtained by a SAT solver, which is one possible outcome of the program. To find all possible outcomes, we add the clause representing the negation of the outcome to the model and run the SAT solver again. As long as there are additional possible outcomes, the SAT solver returns another satisfying assignment. We repeat this process iteratively until the model becomes unsatisfiable. As we only add constraints to the model, the SAT solver uses conflict clauses from previous runs to make subsequent iterations faster. For the prototype, we used minisat [11] as the SAT solver.

Table 1. Contrasting axiomatic and operational models: time/instructions/threads

Operational / Axiomatic	SC	TSO	PSO	RMO	NTSO	NPSO
SC	-	1s/4/2	1s/4/2	1s/4/2	8s/4/2	1s/4/2
TSO	1s/4/2	-	1s/4/2	1s/4/2	130s/5/3	1s/4/2
PSO	1s/4/2	1s/4/2	-	1s/4/2	8s/4/2	16s/5/3
RMO	1s/4/2	1s/4/2	1s/4/2	-	8s/4/2	16s/5/3
NTSO	2s/4/2	39s/5/3	2s/4/2	2s/4/2	-	2s/4/2
NPSO	2s/4/2	2s/4/2	40s/5/3	2s/4/2	9s/4/2	-

4 Experiments

This section describes the experiments we performed to demonstrate the feasibility and usefulness of our approach, including: (1) measuring the execution time for contrasting the operational and axiomatic specifications of six memory models, (2) showing the effectiveness of the reductions targeted at reducing the number of test programs considered, and (3) performing two case studies in which the tool is used to debug memory model specifications.

4.1 Comparing Different Memory Models

We tested our technique by comparing the operational and axiomatic specifications for various memory models: SC, the three SPARC memory models, and the non-store-atomic extensions of TSO and PSO. As seen in Table 1, a counter example is found for most cases within less than a second. The slowest times occur when comparing models to their non-store-atomic extension, which takes over two minutes for TSO versus NTSO. The litmus tests produced by the tool as counter examples were mostly the litmus tests we expected. However, the tool found subtle errors in our initial operational specification for RMO and for NTSO, which we fixed.

4.2 Test Reductions and Scalability

The graph in Fig. 5 shows the number of tests generated with up to three memory locations, up to three instructions per thread, and a varying number of total instructions. Fences are not counted towards the total number of instructions. Symmetry reductions provide approximately a 10x reduction in the number of tests, and redundant program elimination provides an additional 10x reduction, resulting in an overall reduction by a factor of 100x in the number of generated tests. The graph in Fig. 6 shows the average time per test for both operational and axiomatic memory models. As seen in this graph, the average time per test is no more than several seconds for programs with up to nine instructions, which means a bound of six or seven instructions can be explored in a reasonable time.

4.3 Debugging Our Axiomatic Specification for NPSO

As a case study for using our technique for debugging a new memory model specification, we developed an axiomatic specification for NPSO, a non-store-atomic

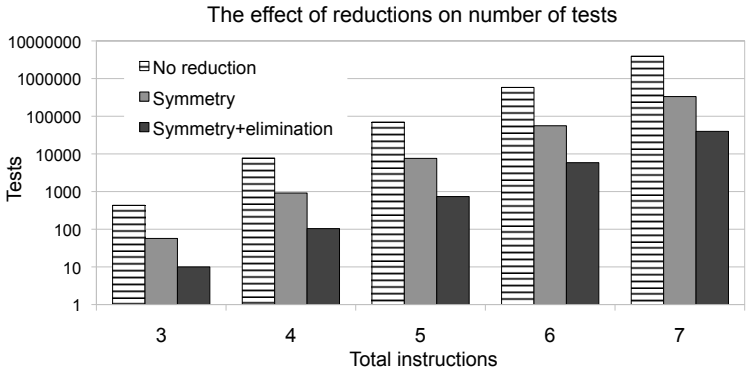


Fig. 5. The effect of reductions on the number of tests

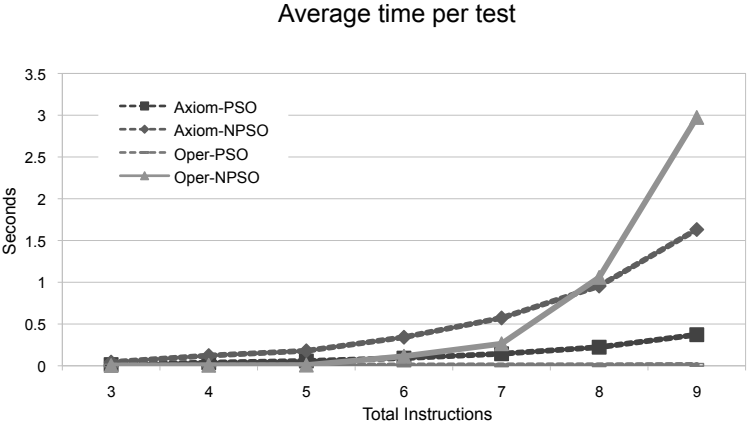


Fig. 6. Average run time per test

relaxation of PSO. We used an existing operational specification for NPSO as a reference model. We started with the axiomatic specification defined in Section 2.2, which is an extension of PSO that allows each thread to observe memory operations in a different order with the addition of a coherence axiom. We then ran the prototype with a bound of six instructions.

The prototype reported that Test G in Fig. 7 is allowed in the axiomatic but not in the operational specification. This is a well-known litmus test, which usually illustrates reorderings of reads after later writes. In this specification, however, we explicitly disallow reordering reads after writes. This outcome occurred because threads are not required to agree on the order of writes to different addresses. To correct the specification, we must rule out this kind of behavior and enforce some notion of causal transitivity. Our first attempt to fix it required that if a read sees a write to the same address in some thread, it can be ordered only after this read in the local thread that issued the write. Running the tool

Test G

T1	T2
Read $X \rightarrow r1$	Read $Y \rightarrow r2$
Write $Y \leftarrow 1$	Write $X \leftarrow 2$

Outcome: $r1 = 2$; $r2 = 1$
time to find: 2s

Test H

T1	T2	T3
Read $X \rightarrow r1$	Read $Y \rightarrow r2$	Read $Z \rightarrow r3$
Write $Y \leftarrow 1$	Write $Z \leftarrow 2$	Write $X \leftarrow 3$

Outcome: $r1 = 3$; $r2 = 1$; $r3 = 2$
time to find: 824s

Test I

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 2$
Fence	Fence
Read $Y \rightarrow r1$	Read $X \rightarrow r2$

Outcome: $r1 = 0$; $r2 = 0$
time to find: 22s

Test J

T1	T2	T3
Write $X \leftarrow 1$	Read $Y \rightarrow r2$	Write $Y \leftarrow 2$
Fence	Read $X \rightarrow r3$	
Write $Y \leftarrow 2$		

Outcome: $r1 = 0$; $r2 = 2$; $r3 = 0$
time to find: 411s

Fig. 7. Litmus tests generated for buggy NPSO specifications

Test K

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 3$
Write $Y \leftarrow 2$	Read $Y \rightarrow r2$
Read $Y \rightarrow r1$	Read $X \rightarrow r3$

Outcome: $r1 = 3$; $r2 = 3$; $r3 = 0$
time to find: 111s

Test L

T1	T2
Write $X \leftarrow 1$	Write $Y \leftarrow 2$
Fence	Read $Y \rightarrow r2$
Read $Y \rightarrow r1$	Read $X \rightarrow r3$

Outcome: $r1 = 0$; $r2 = 2$; $r3 = 0$
time to find: 43s

Fig. 8. Litmus tests generated for SOBER

again after this modification generated Test H in Fig. 7. The proposed axiom was sufficient to rule out cycles involving two threads, but not cycles involving three threads and three addresses. We fixed this by using an alternative axiom, stating that if a read precedes a write to any address according to the local thread of this write, it will precede this write in any other thread.

After fixing the issue of causal transitivity, we ran the prototype again and received Test I in Fig. 7. This outcome is allowed when fences affect only local order and there is no total order among fences. We fixed it by adding an axiom that requires a total order between fences. In the final iteration, we received Test J in Fig. 7. In this case, the operational model drains both the local and the global queues after a fence, which rules out the outcome listed under Test J. A total order between fences is not sufficient to rule out this outcome. We strengthen the total order axiom by requiring all threads to agree about the order between fences and any other operations. After fixing this axiom, we found no new mismatches between the models.

4.4 Debugging the Axiomatic Specification of TSO Used in SOBER

The second case study for our technique was debugging the axiomatic specification of TSO used by SOBER [5]. SOBER is a technique for detecting potential SC violations in software. SOBER uses an axiomatically defined memory model

that is intended to be equivalent to SPARC's TSO. The authors stated that their axiomatic definition is equivalent to their operational specification of TSO [6]. However, Burnim et al [7] discovered that SOBER's axiomatic specification and TSO are, in fact, not equivalent. We used SOBER's specification as a case study to see if our technique could detect the discrepancy between the two models without any prior knowledge about the nature of this discrepancy.

We compared SOBER's axiomatic specification with our operational specification for TSO. Our tool took less than two minutes to generate Test K in Fig. 8, which is allowed by TSO but not by SOBER's specification. Such a test is often used to distinguish TSO from IBM 370 [1], which is essentially TSO without forwarding. We then contrasted SOBER with IBM 370 and received Test L in Fig. 8, demonstrating that SOBER allows behaviors that are not allowed by IBM 370. We implemented a fix suggest by Burckhardt (personal communication), and we found no new mismatch between the fixed model our specification of TSO.

5 Related Work

Many studies describe tools for testing litmus tests on a formally specified memory model [10,20,21,23,24]. Given a parallel program and an expected outcome, these tools report whether the specified outcome is feasible on a specified memory model. Most of these tools test for one outcome at a time [10,20,23,24]. Park and Dill [21] presented a tool that enabled exploring all outcomes for a given parallel program using an operational specification for RMO.

Another approach for debugging a memory model is the "test model-checking" methodology [19]. In this approach, a memory model is verified against a state machine that generates a non-deterministic sequence of writes and test for certain assertions. Each test-generating state machine is designed to detect a certain architectural rule. This approach provides a stronger verification than testing specific litmus tests.

A technique for validating that a system correctly implements a memory model is dynamic testing, which is used by tools such as TSOtool [13] and LCHECK [9]. These tools generate random tests, execute them on a certain hardware, and verify that the execution adheres to a given memory model.

Few studies involve a direct comparison between two memory models. Chatterjee et al [8] shows the equivalence of an operational specification of the Alpha memory model to an implementation of the same model. This work finds a refinement map between the two models via model-checking and uses an intermediate abstraction that exploit structural similarities between the two models to facilitate the proof. Other studies [10,20] use theorem proving to prove equivalence between an operational and axiomatic specification of the same model.

6 Conclusions

We presented a technique for contrasting memory models and implemented a prototype based on this technique. Our experiments showed that this approach

can detect differences between memory models within seconds or minutes, and the case studies showed that by contrasting memory models we can detect subtle differences between memory models that might have gone undetected using a predetermined set of litmus tests. Several key features make this technique a viable tool for debugging memory model specifications: it provides feedback in reasonable time, it generates a minimal-length litmus test as a counter example, which are easy to analyze and understand, it is fully automatic, and it is flexible and general in the sense that it can support different memory models, specification styles, and exploration techniques.

One limitation of our approach is that it does not provide a complete verification for the equivalence of two models. We test programs only up to a certain bound, and we cannot guarantee that there is no longer test that differentiates between the two specifications. Furthermore, redundant program elimination reductions may not be safe when comparing some models. We plan to extend this work to equivalence verification by finding sufficient bounds for a rich but restricted domain of memory models and prove that the reductions we use are safe for this domain of models.

Acknowledgements

We thank Sebastian Burckhardt for suggesting the use of SOBER's TSO specification as a case study for this paper.

References

1. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29, 66–76 (1996)
2. Alglave, J., Fox, A., Ishtiaq, S., Myreen, M.O., Sarkar, S., Sewell, P., Nardelli, F.Z.: The semantics of power and ARM multiprocessor machine code. In: DAMP (2009)
3. Boehm, H.J., Adve, S.V.: Foundations of the C++ concurrency memory model. In: PLDI, pp. 68–78 (2008)
4. Burckhardt, S., Alur, R., Martin, M.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
5. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 107–120. Springer, Heidelberg (2008)
6. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. Technical Report MSR-TR-2008-12, Microsoft Research (2008)
7. Burnim, J., Sen, K., Stergiou, C.: Sound and complete monitoring of sequential consistency in relaxed memory models. Technical Report UCB/EECS-2010-31, EECS Department, University of California, Berkeley (March 2010)
8. Chatterjee, P., Sivaraj, H., Gopalakrishnan, G.: Shared memory consistency protocol verification against weak memory models: Refinement via model-checking. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 123–136. Springer, Heidelberg (2002)
9. Chen, Y., Lv, Y., Hu, W., Chen, T., Shen, H., Wang, P., Pan, H.: Fast complete memory consistency verification. In: HPCA, pp. 381–392 (2009)

10. Chong, N., Ishtiaq, S.: Reasoning about the ARM weakly consistent memory model. In: MSPC, pp. 16–19. ACM, New York (2008)
11. Een, N., Sorensson, N.: Minisat - a SAT solver with conflict-clause minimization. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569. Springer, Heidelberg (2005)
12. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem. Springer, Heidelberg (1996)
13. Hangal, S., Vahia, D., Manovit, C., Lu, J.Y.J.: TSOtool: A program for verifying memory systems using the memory consistency model. ISCA 32(2), 114 (2004)
14. Holzmann, G.J.: The model checker spin. IEEE Transactions on Software Engineering 23, 279–295 (1997)
15. Intel Corporation: Intel 64 and IA-32 Architectures Software Developer's Manual (March 2010)
16. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers 28(9), 690–691 (1979)
17. Mador-Haim, S., Alur, R., Martin, M.: Generating litmus tests for contrasting memory consistency models - extended version. Technical report, Dept. of Computer Information Science, U. of Pennsylvania (2010)
18. Manson, J., Pugh, W., Adve, S.V.: The Java memory model. In: POPL, pp. 378–391 (2005)
19. Nalumasu, R., Ghughal, R., Mokkedem, A., Gopalakrishnan, G.: The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 464–476. Springer, Heidelberg (1998)
20. Owens, S., Sarkar, S., Sewell, P.: A better x86 memory model: x86-TSO. In: TPHOLs, pp. 391–407 (2009)
21. Park, S., Dill, D.L.: An executable specification and verifier for relaxed memory order. IEEE Transactions on Computers 48 (1999)
22. Weaver, D.L., Germond, T.: The SPARC Architecture Manual Version 9. Prentice Hall PTR, Englewood Cliffs (1994)
23. Yang, Y., Gopalakrishnan, G., Lindstrom, G.: UMM: an operational memory model specification framework with integrated model checking capability. Concurr. Comput.: Pract. Exper. 17(5-6), 465–487 (2005)
24. Yang, Y., Gopalakrishnan, G., Lindstrom, G., Slind, K.: Analyzing the intel itanium memory ordering rules using logic programming and SAT. In: Geist, D., Tronci, E. (eds.) CHARME 2003. LNCS, vol. 2860, pp. 81–95. Springer, Heidelberg (2003)

Directed Proof Generation for Machine Code*

Aditya Thakur[1], Junghee Lim[1], Akash Lal[2], Amanda Burton[1], Evan Driscoll[1], Matt Elder[1,**], Tycho Andersen[1], and Thomas Reps[1,3,***]

[1] University of Wisconsin; Madison, WI, USA
[2] Microsoft Research India; Bangalore, India
[3] GrammaTech, Inc.; Ithaca, NY, USA

Abstract. We present the algorithms used in McVeto (**M**achine-**C**ode **VE**rification **TO**ol), a tool to check whether a stripped machine-code program satisfies a safety property. The verification problem that McVeto addresses is challenging because it cannot assume that it has access to (i) certain structures commonly relied on by source-code verification tools, such as control-flow graphs and call-graphs, and (ii) meta-data, such as information about variables, types, and aliasing. It cannot even rely on out-of-scope local variables and return addresses being protected from the program's actions. What distinguishes McVeto from other work on software model checking is that it shows how verification of machine-code can be performed, while avoiding conventional techniques that would be unsound if applied at the machine-code level.

1 Introduction

Recent research has led to new kinds of tools for analyzing programs for bugs and security vulnerabilities. In these tools, program analysis conservatively answers the question "Can the program reach a bad state?" Many impressive results have been achieved; however, the vast majority of existing tools analyze source code, whereas most programs are delivered as machine code. If analysts wish to vet such programs for bugs and security vulnerabilities, tools for analyzing machine code are needed.

Machine-code analysis presents many new challenges. For instance, at the machine-code level, memory is one large byte-addressable array, and an analyzer must handle computed—and possibly non-aligned—addresses. It is crucial to track array accesses and updates accurately; however, the task is complicated by the fact that arithmetic and dereferencing operations are both pervasive and inextricably intermingled. For instance, if local variable `x` is at offset –12 from the activation record's frame pointer (register `ebp`), an access on `x` would be turned

* Supported, in part, by NSF under grants CCF-{0540955, 0810053, 0904371}, by ONR under grants N00014-{09-1-0510, 09-1-0776}, by ARL under grant W911NF-09-1-0413, and by AFRL under grant FA9550-09-1-0279.
** Supported by an NSF Graduate Fellowship.
*** T. Reps has an ownership interest in GrammaTech, Inc., which has licensed elements of the technology reported in this publication.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 288–305, 2010.

into an operand [ebp–12]. Evaluating the operand first involves pointer arithmetic ("ebp–12") and then dereferencing the computed address ("[·]"). On the other hand, machine-code analysis also offers new opportunities, in particular, the opportunity to track low-level, platform-specific details, such as memory-layout effects. Programmers are typically unaware of such details; however, they are often the source of exploitable security vulnerabilities.

The algorithms used in software model checkers that work on source code [5,15,6] would be be unsound if applied to machine code. For instance, before starting the verification process proper, SLAM [5] and BLAST [15] perform flow-insensitive (and possibly field-sensitive) points-to analysis. However, such analyses often make unsound assumptions, such as assuming that the result of an arithmetic operation on a pointer always remains inside the pointer's original target. Such an approach assumes—without checking—that the program is ANSI C compliant, and hence causes the model checker to ignore behaviors that are allowed by some compilers (e.g., arithmetic is performed on pointers that are subsequently used for indirect function calls; pointers move off the ends of structs or arrays, and are subsequently dereferenced). A program can use such features for good reasons—e.g., as a way for a C program to simulate subclassing—but they can also be a source of bugs and security vulnerabilities.

This paper presents the techniques that we have implemented in a model checker for machine code, called MCVETO (**M**achine-**C**ode **VE**rification **TO**ol). MCVETO uses *directed proof generation* (DPG) [13] to find either an input that causes a (bad) target state to be reached, or a proof that the bad state cannot be reached. (The third possibility is that MCVETO fails to terminate.)

What distinguishes the work on MCVETO is that it addresses a large number of issues that have been ignored in previous work on software model checking, and would cause previous techniques to be unsound if applied to machine code. The contributions of our work can be summarized as follows:

1. We show how to verify safety properties of machine code while avoiding a host of assumptions that are unsound in general, and that would be inappropriate in the machine-code context, such as reliance on symbol-table, debugging, or type information, and preprocessing steps for (a) building a precomputed, fixed, interprocedural control-flow graph (ICFG), or (b) performing points-to/alias analysis.
2. MCVETO builds its (sound) abstraction of the program's state space on-the-fly, performing disassembly one instruction at a time during state-space exploration, without static knowledge of the split between code vs. data. (It does not have to be prepared to disassemble *collections* of nested branches, loops, procedures, or the whole program all at once, which is what can confuse conventional disassemblers [20].)

   The initial abstraction has only two abstract states, defined by the predicates "PC = *target*" and "PC ≠ *target*" (where "PC" denotes the program counter). The abstraction is gradually refined as more of the program is

exercised (§3). MCVETO can analyze programs with instruction aliasing[1] because it builds its abstraction of the program's state space entirely on-the-fly (§3.1). Moreover, MCVETO is capable of verifying (or detecting flaws in) self-modifying code (SMC). With SMC there is no fixed association between an address and the instruction at that address, but this is handled automatically by MCVETO's mechanisms for abstraction refinement. To the best of our knowledge, MCVETO is the first model checker to handle SMC.

3. MCVETO introduces *trace generalization*, a new technique for eliminating *families* of infeasible traces (§3.1). Compared to prior techniques that also have this ability [14], our technique involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation.*
4. MCVETO introduces a new approach to performing DPG on multi-procedure programs (§3.3). Godefroid et al. [12] presented a declarative framework that codifies the mechanisms used for DPG in SYNERGY [13], DASH [6], and SMASH [12] (which are all instances of the framework). In their framework, *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, MCVETO's algorithm lies outside of that framework: the interprocedural component of MCVETO uses (and refines) an *infinite graph*, which is finitely represented and queried by *symbolic operations.*
5. We developed a language-independent algorithm to identify the aliasing condition relevant to a property in a given state (§3.4). Unlike previous techniques [6], it applies when static names for variables/objects are unavailable.

Items 1 and 2 address execution details that are typically ignored (unsoundly) by source-code analyzers. Items 3, 4, and 5 are applicable to both source-code and machine-code analysis. MCVETO is not restricted to an impoverished language. In particular, it handles pointers and bit-vector arithmetic.

Organization. §2 contains a brief review of DPG. §3 describes the new DPG techniques used in MCVETO. §4 describes how different instances of MCVETO are generated automatically from a specification of the semantics of an instruction set. §5 presents experimental results. §6 discusses related work. §7 concludes.

2 Background on Directed Proof Generation (DPG)

Given a program P and a particular control location *target* in P, DPG returns either an input for which execution leads to *target* or a proof that *target* is unreachable (or DPG does not terminate). Two approximations of P's state space are maintained:

- A set T of concrete traces, obtained by running P with specific inputs. T *under*approximates P's state space.
- A graph G, called the *abstract graph*, obtained from P via abstraction (and abstraction refinement). G *over*approximates P's state space.

[1] Programs written in instruction sets with varying-length instructions, such as x86, can have "hidden" instructions starting at positions that are out of registration with the instruction boundaries of a given reading of an instruction stream [20].

Nodes in G are labeled with formulas; edges are labeled with program statements or program conditions. One node is the *start node* (where execution begins); another node is the *target node* (the goal to reach). Information to relate the under- and overapproximations is also maintained: a concrete state σ in a trace in T is called a *witness* for a node n in G if σ satisfies the formula that labels n.

If G has no path from *start* to *target*, then DPG has proved that *target* is unreachable, and G serves as the proof. Otherwise, DPG locates a *frontier*: a triple (n, I, m), where (n, m) is an edge on a path from *start* to *target* such that n has a witness w but m does not, and I is the instruction on (n, m). DPG either performs concrete execution (attempting to reach *target*) or refines G by splitting nodes and removing certain edges (which may prove that *target* is unreachable). Which action to perform is determined using the basic step from *directed test generation* [10], which uses symbolic execution to try to find an input that allows execution to cross frontier (n, I, m). Symbolic execution is performed over symbolic states, which have two components: a *path constraint*, which represents a constraint on the input state, and a *symbolic map*, which represents the current state in terms of input-state quantities. DPG performs symbolic execution along the path taken during the concrete execution that produced witness w for n; it then symbolically executes I, and conjoins to the path constraint the formula obtained by evaluating m's predicate ψ with respect to the symbolic map. It calls an SMT solver to determine if the path constraint obtained in this way is satisfiable. If so, the result is a satisfying assignment that is used to add a new execution trace to T. If not, DPG refines G by splitting node n into n' and n'', as shown in Fig. 1.

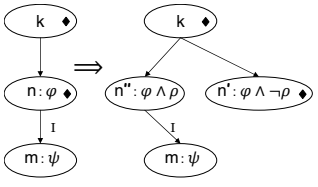


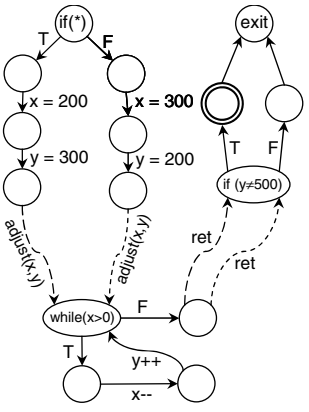
Fig. 1. The general refinement step across frontier (n, I, m). The presence of a witness is indicated by a "♦" inside of a node.

Refinement changes G to represent some *non-connectivity* information: in particular, n' is not connected to m in the refined graph (see Fig. 1). Let ψ be the formula that labels m, c be the concrete witness of n, and S_n be the symbolic state obtained from the symbolic execution up to n. DPG chooses a formula ρ, called the *refinement predicate*, and splits node n into n' and n'' to distinguish the cases when n is reached with a concrete state that satisfies ρ (n'') and when it is reached with a state that satisfies $\neg\rho$ (n'). The predicate ρ is chosen such that (i) no state that satisfies $\neg\rho$ can lead to a state that satisfies ψ after the execution of I, and (ii) the symbolic state S_n satisfies $\neg\rho$. Condition (i) ensures that the edge from n' to m can be removed. Condition (ii) prohibits extending the current path along I (forcing the DPG search to explore different paths). It also ensures that c is a witness for n' and not for n'' (because c satisfies S_n)—and thus the frontier during the next iteration must be different.

```
int main() {
  int x, y;
  if(*) {
    x = 200;
    y = 300;
    adjust(x, y);
  }
  else {
    x = 300;
    y = 200;
    adjust(x, y);
  }
  if (y != 500)
    UNREACHABLE();
  return 0;
}
```

(a)

```
void adjust(int& x, int& y) {
  while(x > 0) {
    x--;
    y++;
  }
}
```



(b)

Fig. 2. (a) A program with a non-deterministic branch; (b) the program's ICFG

3 McVeto

This section explains the methods used to achieve contributions 1–5 listed in §1. While McVeto was designed to provide sound DPG for machine code, a number of its novel features are also useful for source-code DPG. Thus, to make the paper more accessible, our running example is the C++ program in Fig. 2. It makes a non-deterministic choice between two blocks that each call procedure `adjust`, which loops—decrementing `x` and incrementing `y`. Note that the affine relation $x + y = 500$ holds at the two calls on `adjust`, the loop-head in `adjust`, and the branch on `y!=500`.

Representing the Abstract Graph. The infinite abstract graph used in McVeto is finitely represented as a nested word automaton (NWA) [2] and queried by symbolic operations. As discussed in §3.1 the key property of NWAs for abstraction refinement is that, even though they represent matched call/return structure, they are closed under intersection [2]. That is, given NWAs A_1 and A_2, one can construct an NWA A_3 such that $L(A_3) = L(A_1) \cap L(A_2)$.

In our NWAs, the alphabet consists of all possible machine-code instructions. In addition, we annotate each state with a predicate. Operations on NWAs extend cleanly to accommodate the semantics of predicates—e.g., the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction of the predicates on states q_1 and q_2. In McVeto's abstract graph, we treat the value of the PC as data; consequently, predicates can refer to the value of the PC (see Fig. 3).

3.1 Abstraction Refinement via Trace Generalization

In a source-code model checker, the initial overapproximation of a program's state space is often the program's ICFG. Unfortunately, for machine code it is difficult to create an accurate ICFG *a priori* because of the use of indirect

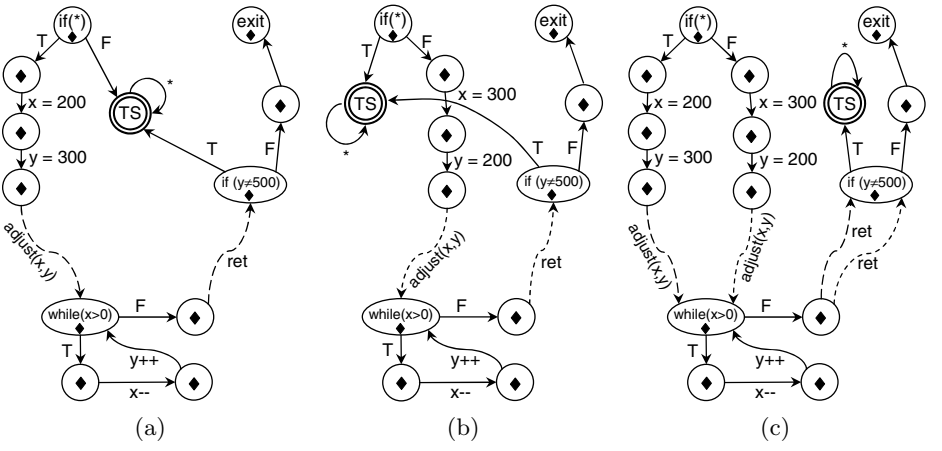


Fig. 4. (a) and (b) Two generalized traces, each of which reaches the end of the program. (c) The intersection of the two generalized traces. (A "♦" indicates that a node has a witness.)

jumps, jump tables, and indirect function calls—as well as more esoteric features, such as instruction aliasing and SMC. For this reason, MCVETO begins with the degenerate NWA-based abstract graph G_0 shown in Fig. 3, which overapproximates the program's state space; i.e., G_o accepts an overapproximation of the set of minimal[2] traces that reach *target*. The abstract graph is refined during the state-space exploration carried out by MCVETO.

To avoid having to disassemble collections of nested branches, loops, procedures, or the whole program all at once, MCVETO performs *trace-based disassembly*: as concrete traces are generated during DPG, instructions are disassembled one at a time by decoding the current bytes of memory starting at the value of the PC. Each indirect jump or indirect call encountered can be resolved to a specific address. Trace-based disassembly is one of the techniques that allows MCVETO to handle self-modifying code.

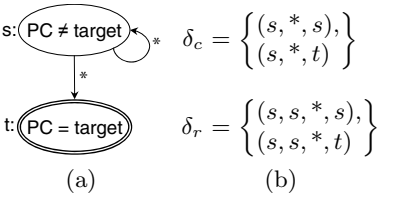


Fig. 3. (a) Internal-transitions in the initial NWA-based abstract graph G_0 created by MCVETO; (b) call- and return-transitions in G_0. * is a wild-card symbol that matches all instructions.

MCVETO uses each concrete trace $\pi \in T$ to refine abstract graph G. As mentioned in §2, the set T of concrete traces *under*approximates the program's state space, whereas G represents an *over*approximation of the state space. MCVETO repeatedly solves instances of the following *trace-generalization* problem:

[2] A trace τ that reaches *target* is *minimal* if τ does not have a proper prefix that reaches *target*.

Given a trace π, which is an *under*approximation of the program, convert π into an NWA-based abstract graph G_π that is an *over*approximation of the program.

We create G_π by "folding" π—grouping together all nodes with the same PC value, and augmenting it in a way that overapproximates the portion of the program not explored by π (denoted by $\pi/[\text{PC}]$); see Figs. 4(a) and (b) and Fig. 5. In particular, G_π contains one accepting state, called *TS* (for "target surrogate"). *TS* is an accepting state because it represents *target*, as well as all non-*target* locations not visited by π. (Trace generalization for SMC is discussed in [25, §3.1].)

We now have two overapproximations, the original abstract graph G and folded trace G_π. Thus, by performing $G := G \cap G_\pi$, information about the portion of the program explored by π is incorporated into G, producing a third, improved overapproximation; see Fig. 4(c). (Equivalently, intersection eliminates the family of infeasible traces represented by the complement of G_π; however, because we already have G_π in hand, no automaton-complementation operation is required—cf. [14].)

The issue of how one forms an NWPrefix from an instruction sequence—i.e., identifying the nesting structure—is handled by a policy in the trace-recovery tool for classifying each position as an internal-, call-, or return-position. Currently, for reasons discussed in §3.5, we use the following policy: the position of any form of `call` instruction is a call-position; the position of any form of `ret` instruction is a return-position. In essence, MCVETO uses `call` and `ret` instructions to restrict the instruction sequences considered. If these match the program's actual instruction sequences, we obtain the benefits of the NWA-based approach—especially the reuse of information among refinements of a given procedure. The basic MCVETO algorithm is stated as Alg. 1.

Algorithm 1. Basic MCVETO algorithm (including trace-based disassembly)

1: $\pi :=$ nested-word prefix for an execution run on a random initial state
2: $T := \{\pi\}$; $G_\pi := \pi/[\text{PC}]$; $G :=$ (NWA from Fig. 3) $\cap\ G_\pi$
3: **loop**
4: **if** *target* has a witness in T **then return** "reachable"
5: Find a path τ in G from *start* to *target*
6: **if** no path exists **then return** "not reachable"
7: Find a frontier (n, I, m) in G, where concrete state σ witnesses n
8: Perform symbolic execution of the instructions of the concrete trace that reaches σ, and then of instruction I; conjoin to the path constraint the formula obtained by evaluating m's predicate ψ with respect to the symbolic map; let S be the path constraint so obtained
9: **if** S is feasible, with satisfying assignment A **then**
10: $\pi :=$ nested-word prefix for an execution run on A
11: $T := T \cup \{\pi\}$; $G_\pi := \pi/[\text{PC}]$; $G := G \cap G_\pi$
12: **else**
13: Refine G along frontier (n, I, m) (see Fig. 1)

Definition 1. *A trace π that does not reach target is represented by (i) a nested-word prefix $(w, \rightsquigarrow)$ over instructions ([25, App. A]), together with (ii) an array of PC values, $PC[1..|w|+1]$, where $PC[|w|+1]$ has the special value HALT if the trace terminated execution.* **Internal-steps**, **call-steps**, *and* **return-steps** *are triples of the form $\langle PC[i], w[i], PC[i+1]\rangle$, $1 \le i < |w|$, depending on whether i is an internal-position, call-position, or return-position, respectively. Given π, we construct $G_\pi \stackrel{\text{def}}{=} \pi/[PC]$ as follows:*

1. *All positions $1 \le k < |w|+1$ for which $PC[k]$ has a given address a are collapsed to a single NWA state q_a. All such states are rejecting states (the target was not reached).*
2. *For each internal-step $\langle a, I, b\rangle$, G_π has an internal-transition (q_a, I, q_b).*
3. *For each call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a call-transition $(q_{a_c}, \texttt{call}, q_{a_e})$. ("`call`" stands for whatever instruction instance was used in the call-step.)*
4. *For each return-step $\langle a_x, \texttt{ret}, a_r\rangle$ for which the PC at the call predecessor holds address a_c, G_π has a return-transition $(q_{a_x}, q_{a_c}, \texttt{ret}, q_{a_r})$. ("`ret`" stands for whatever instruction instance was used in the return-step.)*
5. *G_π contains one accepting state, called TS (for "target surrogate"). TS is an accepting state because it represents target, as well as all the non-target locations that π did not explore.*
6. *G_π contains three "self-loops": $(TS, *, TS) \in \delta_i$, $(TS, *, TS) \in \delta_c$, and $(TS, TS, *, TS) \in \delta_r$. (We use "*" in the latter two transitions because there are many forms of `call` and `ret` instructions.)*
7. *For each unmatched instance of a call-step $\langle a_c, \texttt{call}, a_e\rangle$, G_π has a return-transition $(TS, q_{a_c}, *, TS)$. (We use * because any kind of `ret` instruction could appear in the matching return-step.)*
8. *Let B_b denote a (direct or indirect) branch that takes branch-direction b.*
   - *If π has an internal-step $\langle a, B_b, c\rangle$ but not an internal-step $\langle a, B_{\neg b}, d\rangle$, G_π has an internal-transition $(q_a, B_{\neg b}, TS)$.*
   - *For each internal-step $\langle a, B_T, c\rangle$, where B is an indirect branch, G_π has an internal-transition (q_a, B_T, TS).*
9. *For each call-step $\langle a_c, \texttt{call}, a_e\rangle$ where `call` is an indirect call, G_π has a call-transition $(q_{a_c}, \texttt{call}, TS)$.*
10. *If $PC[|w|+1] \neq HALT$, G_π has an internal-transition $(q_{PC[|w|]}, I, TS)$, where "I" stands for whatever instruction instance was used in step $|w|$ of π. (We assume that an uncompleted trace never stops just before a `call` or `ret`.)*
11. *If $PC[|w|+1] = HALT$, G_π has an internal-transition $(q_{PC[|w|]}, I, Exit)$, where "I" stands for whatever instruction instance was used in step $|w|$ of π and Exit is a distinguished non-accepting state.*

Fig. 5. Definition of the trace-folding operation $\pi/[\text{PC}]$

3.2 Speculative Trace Refinement

Motivated by the observation that DPG is able to avoid exhaustive loop unrolling if it discovers the right loop invariant, we developed mechanisms to discover candidate invariants from a folded trace, which are then incorporated into the abstract graph via NWA intersection. Although they are only *candidate* invariants, they are introduced into the abstract graph in the hope that they are

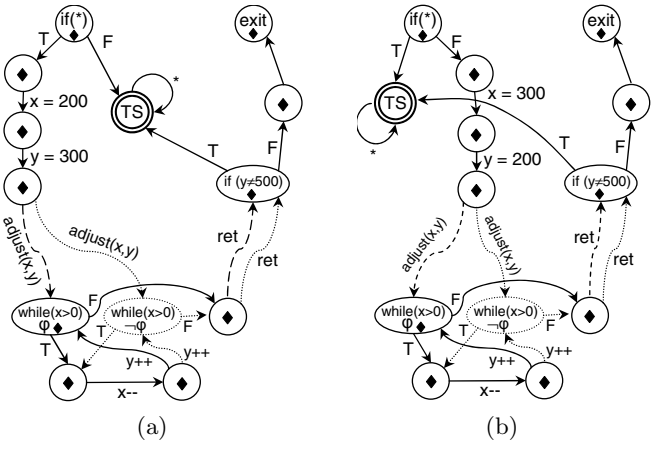


Fig. 6. Fig. 4(a) and (b) with the loop-head in `adjust` split with respect to the candidate invariant $\varphi \stackrel{\text{def}}{=} x + y = 500$

invariants for the full program. The basic idea is to apply dataflow analysis to a graph obtained from the folded trace G_π. The recovery of invariants from G_π is similar in spirit to the computation of invariants from traces in Daikon [9], but in MCVETO they are computed *ex post facto* by dataflow analysis on the folded trace. While any kind of dataflow analysis could be used in this fashion, MCVETO currently uses two analyses:

- Affine-relation analysis [21] is used to obtain linear equalities over registers and a set of memory locations, V. V is computed by running aggregate structure identification [22] on G_π to obtain a set of inferred memory variables M, then selecting $V \subseteq M$ as the most frequently accessed locations in π.
- An analysis based on strided-interval arithmetic [23] is used to discover range and congruence constraints on the values of individual registers and memory locations.

The candidate invariants are used to create predicates for the nodes of G_π. Because an analysis may not account for the full effects of indirect memory references on the inferred variables, to incorporate a discovered candidate invariant φ for node n into G_π safely, we split n on φ and $\neg\varphi$. Again we have two overapproximations: G_π, from the folded trace, augmented with the candidate invariants, and the original abstract graph G. To incorporate the candidate invariants into G, we perform $G := G \cap G_\pi$; the $\cap$ operation labels a product state $\langle q_1, q_2 \rangle$ with the conjunction of the predicates on states q_1 of G and q_2 of G_π.

Fig. 6 shows how the candidate affine relation $\varphi \stackrel{\text{def}}{=} x + y = 500$ would be introduced at the loop-head of `adjust` in the generalized traces from Figs. 4(a) and (b). (Relation φ does, in fact, hold for the portions of the state space explored by Figs. 4(a) and (b).) With this enhancement, subsequent steps of DPG will be able to show that the dotted loop-heads (labeled with $\neg\varphi$) can never be reached

```
int y;
void baz(){
  y=0;
  y++;
  y--;
}
```

```
void lotsaBaz(int a){
  y=0;
  if(a>0) baz();
  if(a>1) baz();
  if(a>2) baz();
  if(a>3) baz();
  if(y!=0)
    ERR: return;
}
```

(a)

```
int bar1() {
  int i,r = 0;
  for(i=0;i<100;i++){
    complicated(); r++;
  }
  return r;
}

int bar2(){ return 10; }
```

```
void foo(int x){
  int y;
  if(x == 0) y = bar2();
  else y = bar1();
  if(y == 10)
    ERR: return;
}
```

(b)

Fig. 7. Programs that illustrate the benefit of using a conceptually infinite abstract graph

from *start*. In addition, the predicate φ on the solid loop-heads enables DPG to avoid exhaustive loop unrolling to show that the true branch of `y!=500` can never be taken.

3.3 Symbolic Methods for Interprocedural DPG

In other DPG systems [13,6,12], *inter*procedural DPG is performed by invoking *intra*procedural DPG as a subroutine. In contrast, MCVETO analyzes a representation of the entire program (refined on-the-fly), which allows it to reuse all information from previous refinement steps. For instance, in the program shown in Fig. 7(a), procedure `lotsaBaz` makes several calls to `baz`. By invoking analysis once for each call site on `baz`, a tool such as DASH has to re-learn that `y` is set to 0. In contrast, MCVETO only needs to learn this once and gets automatic reuse at all call sites. Note that such reuse is achieved in a different way in SMASH [12], which makes use of explicit procedure summaries. However, because the split between local and global variables is not known when analyzing machine code, it is not clear to us how MCVETO could generate such explicit summaries.

Furthermore, SMASH is still restricted to invoking intraprocedural analysis as a subroutine, whereas MCVETO is not limited to considering frontiers in just a single procedure: at each stage, it is free to choose a frontier in *any* procedure. To see why such freedom can be important, consider the example in Fig. 7(b) (where *target* is `ERR`). DASH might proceed as follows. The initial test uses $[\mathtt{x} \mapsto 42]$, which goes through `bar1`, but does not reach *target*. After a few iterations, the frontier is the call to `bar1`, at which point DASH is invoked on `bar1` to prove that the return value is not 10. The subproof takes a long time because of the complicated loop in `bar1`. In essence, DASH gets stuck in `bar1` without recourse to an easier way to reach *target*. MCVETO can make the same choices, and would start to prove the same property for the return value of `bar1`. However, refinements inside of `bar1` cause the abstract graph to grow, and at some point, if the policy is to pick a frontier *closest* to *target*, the frontier switches to one in `main` that is closer to *target*—in particular, the true branch of the if-condition `x==0`. MCVETO will be able to extend that frontier by running a test with $[\mathtt{x} \mapsto 0]$, which will go through `bar2` and reach *target*. The challenge that we face to support such flexibility is how to select the frontier while accounting for

paths that reflect the nesting structure of calls and returns. As discussed in [25, §3.3], by doing computations via automata, transducers, and pushdown systems, MCVETO can find the set of *all* frontiers, as well as identify the k *closest* frontiers.

3.4 A Language-Independent Approach to Aliasing Relevant to a Property

This section describes how MCVETO identifies—in a language-independent way suitable for use with machine code—the aliasing condition relevant to a property in a given state. There are two challenges to defining an appropriate notion of aliasing condition for use with machine code: (i) `int`-valued and address-valued quantities are indistinguishable at runtime, and (ii) arithmetic on addresses is used extensively.

Suppose that the frontier is (n, I, m), ψ is the formula on m, and S_n is the symbolic state obtained via symbolic execution of a concrete trace that reaches n. For source code, Beckman et al. [6] identify aliasing condition α by looking at the relationship, in S_n, between the addresses written to by I and the ones used in ψ. However, their algorithm for computing α is language-*dependent*: their algorithm has the semantics of C implicitly encoded in its search for "the addresses written to by I". In contrast, as explained below, we developed an alternative, language-*independent* approach, both to identifying α and computing Pre_α.

To simplify the discussion, suppose that a concrete machine-code state is represented using two maps $M : \mathit{INT} \to \mathit{INT}$ and $R : \mathit{REG} \to \mathit{INT}$. Map M represents memory, and map R represents the values of machine registers. We use the standard theory of arrays to describe (functional) updates and accesses on maps, e.g., $\mathit{update}(m, k, d)$ denotes the map m with index k updated with the value d, and $\mathit{access}(m, k)$ is the value stored at index k in m. (We use the notation $m(r)$ as a shorthand for $\mathit{access}(m, r)$.) We also use the standard axiom from the theory of arrays: $(\mathit{update}(m, k_1, d))(k_2) = \mathit{ite}(k_1 = k_2, d, m(k_2))$, where *ite* is an *if-then-else* term. Suppose that I is "`mov [eax],5`" (which corresponds to `*eax = 5` in source-code notation) and that ψ is $(M(R(\texttt{ebp}) - 8) + M(R(\texttt{ebp}) - 12) = 10)$.[3] First, we symbolically execute I starting from the identity symbolic state $S_{id} = [M \mapsto M, R \mapsto R]$ to obtain the symbolic state $S' = [M \mapsto \mathit{update}(M, R(\texttt{eax}), 5), R \mapsto R]$. Next, we evaluate ψ under S'—i.e., perform the substitution $\psi[M \leftarrow S'(M), R \leftarrow S'(R)]$. For instance, the term $M(R(\texttt{ebp}) - 8)$, which denotes the contents of memory at address $R(\texttt{ebp}) - 8$, evaluates to $(\mathit{update}(M, R(\texttt{eax}), 5))(R(\texttt{ebp}) - 8)$. From the axiom for arrays, this simplifies to $\mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8))$. Thus, the evaluation of ψ under S' yields

$$\begin{pmatrix} \mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 8, 5, M(R(\texttt{ebp}) - 8)) \\ + \ \mathit{ite}(R(\texttt{eax}) = R(\texttt{ebp}) - 12, 5, M(R(\texttt{ebp}) - 12)) \end{pmatrix} = 10 \qquad (1)$$

This formula equals $\text{Pre}(I, \psi)$ [18].

[3] In x86, `ebp` is the frame pointer, so if program variable `x` is at offset –8 and `y` is at offset –12, ψ corresponds to $\texttt{x} + \texttt{y} = 10$.

The process described above illustrates a general property: for any instruction I and formula ψ, $\mathrm{Pre}(I,\psi) = \psi[M \leftarrow S'(M), R \leftarrow S'(R)]$, where $S' = \mathrm{SE}[\![I]\!]S_{id}$ and $\mathrm{SE}[\![\cdot]\!]$ denotes symbolic execution [18].

The next steps are to identify α and to create a simplified formula ψ' that weakens $\mathrm{Pre}(I,\psi)$. These are carried out simultaneously during a traversal of $\mathrm{Pre}(I,\psi)$ that makes use of the symbolic state S_n at node n. We illustrate this on the example discussed above for a case in which $S_n(R) = [\texttt{eax} \mapsto R(\texttt{ebp}) - 8]$ (i.e., continuing the scenario from footnote 3, `eax` holds `&x`). Because the *ite*-terms in Eqn. (1) were generated from array accesses, *ite*-conditions represent possible constituents of aliasing conditions. We initialize α to *true* and traverse Eqn. (1). For each subterm t of the form $ite(\varphi, t_1, t_2)$ where φ definitely holds in symbolic state S_n, t is simplified to t_1 and φ is conjoined to α. If φ can never hold in S_n, t is simplified to t_2 and $\neg\varphi$ is conjoined to α. If φ can sometimes hold and sometimes fail to hold in S_n, t and α are left unchanged.

In our example, $R(\texttt{eax})$ equals $R(\texttt{ebp})-8$ in symbolic state S_n; hence, applying the process described above to Eqn. (1) yields

$$\begin{aligned} \psi' &= (5 + M(R(\texttt{ebp}) - 12) = 10) \\ \alpha &= (R(\texttt{eax}) = R(\texttt{ebp}) - 8) \wedge (R(\texttt{eax}) \neq R(\texttt{ebp}) - 12) \end{aligned} \tag{2}$$

The formula $\alpha \Rightarrow \psi'$ is the desired refinement predicate $\mathrm{Pre}_\alpha(I,\psi)$.

In practice, we found it beneficial to use an alternative approach, which is to perform the same process of evaluating conditions of *ite* terms in $\mathrm{Pre}(I,\psi)$, but to use one of the concrete witness states W_n of frontier node n in place of symbolic state S_n. The latter method is less expensive (it uses formula-evaluation steps in place of SMT solver calls), but generates an aliasing condition specific to W_n rather than one that covers all concrete states described by S_n.

Both approaches are *language-independent* because they isolate where the instruction-set semantics comes into play in $\mathrm{Pre}(I,\psi)$ to the computation of $S' = \mathrm{SE}[\![I]\!]S_{id}$; all remaining steps involve only purely logical primitives. Although our algorithm computes $\mathrm{Pre}(I,\psi)$ explicitly, that step alone does not cause an explosion in formula size; explosion is due to *repeated* application of Pre. In our approach, the formula obtained via $\mathrm{Pre}(I,\psi)$ is immediately simplified to create first ψ', and then $\alpha \Rightarrow \psi'$.

Byte-Addressable Memory. When memory is byte-addressable, the actual memory-map type is *INT32* $\rightarrow$ *INT8*. This complicates matters because accessing (updating) a 32-bit quantity in memory translates into four contiguous 8-bit accesses (updates). [25, §3.4] shows how a result equivalent to Eqn. (2) is obtained for a memory-map of type *INT32* $\rightarrow$ *INT8*.

3.5 Soundness Guarantee

The soundness argument for MCVETO is more subtle than it otherwise might appear because of examples like the one shown in Fig. 8. The statement `*(&r+2)=r;` overwrites `foo`'s return address, and `MakeChoice` returns a random 32-bit number. At the end of `foo`, half the runs return normally to `main`. For the other half,

the `ret` instruction at the end of `foo` serves to call `bar`. The problem is that for a run that returns normally to `main` after trace generalization and intersection with G_0, there is no frontier. Consequently, half of the runs of MCVETO, on average, would erroneously report that location `ERR` is unreachable.

MCVETO uses the following policy P for classifying execution steps: (a) the position of any form of `call` instruction is a call-position; (b) the position of any form of `ret` instruction is a return-position. Our goals are (i) to define a property Q that is compatible with P in the sense that MCVETO can check for violations of Q *while checking only NW-Prefix paths* ([25, App. A]), and (ii) to establish a soundness guarantee: either MCVETO reports that Q is violated (along with an input that demonstrates it), or it reports that *target* is reachable (again with an input that demonstrates it), or it correctly reports that Q is invariant and *target* is unreachable. To define Q, we augment the instruction-set semantics with an auxiliary stack. Initially, the auxiliary stack is empty; at each `call`, a copy of the return address pushed on the processor stack is also pushed on the auxiliary stack; at each `ret`, the auxiliary stack is popped.

```
void bar() {
  ERR: // address here is 0x10
}

void foo() {
  int b = MakeChoice() & 1;
  int r = b*0x68 + (1-b)*0x10;
  *(&r+2) = r;
  return;
}

int main() {
  foo();
  // address here is 0x68
}
```

Fig. 8. `ERR` is reachable, but only along a path in which a `ret` instruction serves to perform a call

Definition 2. *An **acceptable execution** (AE) under the instrumented semantics is one in which at each `ret` instruction (i) the auxiliary stack is non-empty, and (ii) the address popped from the processor stack matches the address popped from the auxiliary stack.*

In the instrumented semantics, a flag V is set whenever the program performs an execution step that violates either condition (i) or (ii) of Defn. 2. Instead of the initial NWA shown in Fig. 3, we use a similar two-state NWA that has states q_1: PC $\neq$ *target* $\wedge \neg$V and q_2: PC $=$ *target* $\vee$ V, where q_1 is non-accepting and q_2 is accepting. In addition, we add one more rule to the trace-generalization construction for G_π from Fig. 5:

12. For each return-step $\langle a_x, \texttt{ret}, a_r \rangle$, G_π has an internal-transition $(q_{a_x}, \texttt{ret}, TS)$.

As shown below, these modifications cause the DPG algorithm to also search for traces that are AE violations.

Theorem 1 (Soundness of MCVETO)

1. *If MCVETO reports "AE violation" (together with an input S), execution of S performs an execution that is not an AE.*
2. *If MCVETO reports "bug found" (together with an input S), execution of S performs an AE to target.*
3. *If MCVETO reports "OK", then (a) the program performs only AEs, and (b) target cannot be reached during any AE.*

Sketch of Proof: If a program has a concrete execution trace that is not AE, there must exist a shortest non-AE prefix, which has the form "NWPrefix `ret`" where either (i) the auxiliary stack is empty, or (ii) the return address used by `ret` from the processor stack fails to match the return address on the auxiliary stack. At each stage, the abstract graph used by MCVETO accepts an overapproximation of the program's shortest non-AE execution-trace prefixes. This is true of the initial graph G_0 because internal transitions have wild-card symbols. Moreover, each folded trace $G_\pi = \pi/[\text{PC}]$ accepts traces of the form "NWPrefix `ret`" due to the addition of internal transitions to *TS* for each `ret` instruction (item 12 above). NWA intersection of two sound overapproximations leads to a refined sound overapproximation. Therefore, when MCVETO has shown that no accepting state is reachable, it has also proved that the program has no AE violations.

4 Implementation

The MCVETO implementation incorporates all of the techniques described in §3. The implementation uses only language-independent techniques; consequently, MCVETO can be easily retargeted to different languages. The main components of MCVETO are language-independent in two different dimensions:

1. The MCVETO DPG driver is structured so that one only needs to provide implementations of primitives for concrete and symbolic execution of a language's constructs, plus a handful of other primitives (e.g., Pre_α). Consequently, this component can be used for both source-level languages and machine-code languages.
2. For machine-code languages, we used two tools that *generate* the required implementations of the primitives for concrete and symbolic execution from descriptions of the syntax and concrete operational semantics of an instruction set. The abstract syntax and concrete semantics are specified using TSL (**T**ransformer **S**pecification **L**anguage) [19]. Translation of binary-encoded instructions to abstract syntax trees is specified using a tool called ISAL (**I**nstruction **S**et **A**rchitecture **L**anguage).

In addition, we developed language-independent solutions to each of the issues in MCVETO, such as identifying the aliasing condition relevant to a specific property in a given state (§3.4). Consequently, our implementation acts as a Yacc-like tool for creating versions of MCVETO for different languages: given a description of language L, a version of MCVETO for L is generated automatically. We created two specific instantiations of MCVETO from descriptions of the Intel x86 and PowerPC instruction sets. To perform symbolic queries on the conceptually-infinite abstract graph ([25, §3.3]), the implementation uses OpenFst [1] (for transducers) and WALi [16] (for WPDSs).

5 Experiments

Our experiments (see Fig. 9) were run on a single core of a single-processor quad-core 3.0 GHz Xeon computer running Windows XP, configured so that a

Program		McVeto performance (x86)				
Name	Outcome	#Instrs	CE	SE	Ref	time
blast2/blast2	timeout	98	**	**	**	**
fib/fib–REACH-0	timeout	49	**	**	**	**
fib/fib–REACH-1	counterex.	49	1	0	0	0.125
slam1/slam1	proof	84	15	129	307	203
smc1/smc1–REACH-0*	proof	21	1	60	188	959
smc1/smc1–REACH-1*	counterex.	21	1	0	0	0.016
ex5/ex	counterex.	48	2	10	38	3.05
doubleloopdep/count–COUNT-6	counterex.	31	7	11	13	11.5
doubleloopdep/count–COUNT-7	counterex.	31	7	11	13	11.6
doubleloopdep/count–COUNT-8	counterex.	31	7	11	13	11.6
doubleloopdep/count–COUNT-9	counterex.	31	7	11	13	11.7
inter.synergy/barber	timeout	253	**	**	**	**
inter.synergy/berkeley	counterex.	104	5	13	16	3.95
inter.synergy/cars	proof	196	11	118	349	188
inter.synergy/efm	timeout	188	**	**	**	**
share/share–CASE-0	proof	50	3	24	75	8.5
cert/underflow	counterex.	120	2	6	12	9.55
instraliasing/instraliasing–REACH-0	proof	46	2	36	126	15.0
instraliasing/instraliasing–REACH-1	counterex.	46	2	17	55	5.86
longjmp/jmp	AE viol.	74	1	0	0	0.015
overview0/overview	proof	49	2	31	91	54.9
small_static_bench/ex5	proof	33	3	7	13	2.27
small_static_bench/ex6	proof	30	1	55	146	153
small_static_bench/ex8	proof	89	4	17	46	6.31
verisec-gxine/simp_bad	counterex.	1067	1	0	0	0.094
verisec-gxine/simp_ok	proof	1068	**	**	**	**
clobber_ret_addr/clobber–CASE-4	AE viol.	43	4	9	18	2.13
clobber_ret_addr/clobber–CASE-8	AE viol.	35	2	2	5	0.625
clobber_ret_addr/clobber–CASE-9	proof	35	1	5	21	1.44

Fig. 9. McVeto experiments. The columns show whether McVeto returned a proof, counterexample, or an AE violation (Outcome); the number of instructions (#Instrs); the number of concrete executions (CE); the number of symbolic executions (SE), which also equals the number of calls to the Yices solver; the number of refinements (Ref), which also equals the number of Pre_α computations; and the total time (in seconds). *SMC test case. **Exceeded twenty-minute time limit.

user process has 4 GB of memory. They were designed to test various aspects of a DPG algorithm and to handle various intricacies that arise in machine code (some of which are not visible in source code). We compiled the programs with Visual Studio 8.0, and ran McVeto on the resulting object files (without using symbol-table information).[4]

The examples `ex5`, `ex6`, and `ex8` are from the NECLA Static Analysis Benchmarks. The examples `barber`, `berkeley`, `cars`, `efm` are multi-procedure versions of the larger examples on which Synergy [13] was tested. (Synergy was tested using single-procedure versions only.) `Instraliasing` illustrates the ability to handle instruction aliasing. (The instruction count for this example was obtained via static disassembly, and hence is only approximate.) `Smc1` illustrates the ability of McVeto to handle self-modifying code. `Underflow` is taken from a DHS tutorial on security vulnerabilities. It illustrates a `strncpy` vulnerability.

[4] The examples are available at `www.cs.wisc.edu/wpis/examples/McVeto`

The examples are small, but challenging. They demonstrate MCVETO's ability to reason automatically about low-level details of machine code using a sequence of sound abstractions. The question of whether the cost of soundness is inherent, or whether there is some way that the well-behavedness of (most) code could be exploited to make the analysis scale better is left for future research.

6 Related Work

Machine-Code Analyzers Targeted at Finding Vulnerabilities. A substantial amount of work exists on techniques to detect security vulnerabilities by analyzing source code. Less work exists on vulnerability detection for machine code. Kruegel et al. [17] developed a system for automating mimicry attacks; it uses symbolic execution to discover attacks that can give up and regain execution control by modifying the contents of the data, heap, or stack so that the application is forced to return control to injected attack code at some point after the execution of a system call. Cova et al. [8] used that platform to detect security vulnerabilities in x86 executables via symbolic execution.

Prior work exists on directed *test* generation for machine code [11,7]. In contrast, MCVETO implements directed *proof* generation. Unlike directed-test-generation tools, MCVETO is goal-directed, and works by trying to refute the claim "no path exists that connects program entry to a given goal state".

Machine-Code Model Checkers. SYNERGY applies to an x86 executable for a "single-procedure C program with only [`int`-valued] variables" [13] (i.e., no pointers). It uses debugging information to obtain information about variables and types, and uses Vulcan [24] to obtain a CFG. It uses integer arithmetic—not bit-vector arithmetic—in its solver. In contrast, MCVETO addresses the challenges of checking properties of stripped executables articulated in §1.

Our group developed two prior machine-code model checkers, CodeSurfer/x86 [4] and DDA/x86 [3]. Neither system uses underapproximation. For overapproximation, both use numeric static analysis and a different form of abstraction refinement.

Trace Generalization. The trace-generalization technique of §3.1 has both similarities to and differences from the *trace-refinement* technique of Heizmann et al. [14]. Both techniques adopt a language-theoretic viewpoint and refine an overapproximation to eliminate *families* of infeasible concrete traces. However, trace generalization obtains the desired outcome in a substantially different way. For Heizmann et al., once a refutation automaton is constructed—which involves calling an SMT solver and an interpolant generator—refinement is performed by automaton complementation followed by automaton intersection. In contrast, our generalized traces are created by generalizing a *feasible concrete trace* to create directly a representation that overapproximates the set of minimal traces that reach *target*. Consequently, refinement by trace generalization involves *no calls on an SMT solver*, and *avoids the potentially expensive step of automaton complementation*.

7 Conclusion

MCVETO resolves many issues that have been unsoundly ignored in previous work on software model checking. MCVETO addresses the challenge of establishing properties of the machine code that actually executes, and thus provides one approach to checking the effects of compilation and optimization on correctness. The contributions of the paper lie in the insights that went into defining the innovations in dynamic and symbolic analysis used in MCVETO: (i) sound disassembly and sound construction of an overapproximation (even in the presence of instruction aliasing and self-modifying code) (§3.1), (ii) a new method to eliminate families of infeasible traces (§3.1), (iii) a method to speculatively, but soundly, elaborate the abstraction in use (§3.2), (iv) new symbolic methods to query the (conceptually infinite) abstract graph (§3.3), and (v) a language-independent approach to Pre_α (§3.4). Not only are our techniques language-independent, the implementation is parameterized by specifications of an instruction set's semantics. By this means, MCVETO has been instantiated for both x86 and PowerPC.

References

1. Allauzen, C., Riley, M., Schalkwyk, J., Skut, W., Mohri, M.: OpenFst: A general and efficient weighted finite-state transducer library. In: Holub, J., Žďárek, J. (eds.) CIAA 2007. LNCS, vol. 4783, pp. 11–23. Springer, Heidelberg (2007)
2. Alur, R., Madhusudan, P.: Adding nesting structure to words. JACM 56 (2009)
3. Balakrishnan, G., Reps, T.: Analyzing stripped device-driver executables. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 124–140. Springer, Heidelberg (2008)
4. Balakrishnan, G., Reps, T., Kidd, N., Lal, A., Lim, J., Melski, D., Gruian, R., Yong, S., Chen, C.-H., Teitelbaum, T.: Model checking x86 executables with CodeSurfer/x86 and WPDS++. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 158–163. Springer, Heidelberg (2005)
5. Ball, T., Rajamani, S.: The SLAM toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, p. 260. Springer, Heidelberg (2001)
6. Beckman, N., Nori, A., Rajamani, S., Simmons, R.: Proofs from tests. In: ISSTA (2008)
7. Brumley, D., Hartwig, C., Liang, Z., Newsome, J., Poosankam, P., Song, D., Yin, H.: Automatically identifying trigger-based behavior in malware. In: Botnet Detection. Springer, Heidelberg (2008)
8. Cova, M., Felmetsger, V., Banks, G., Vigna, G.: Static detection of vulnerabilities in x86 executables. In: Jesshope, C., Egan, C. (eds.) ACSAC 2006. LNCS, vol. 4186. Springer, Heidelberg (2006)
9. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M., Xiao, C.: The Daikon system for dynamic detection of likely invariants. SCP 69 (2007)
10. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed automated random testing. In: PLDI (2005)
11. Godefroid, P., Levin, M., Molnar, D.: Automated whitebox fuzz testing. In: NDSS (2008)

12. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL (2010)
13. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: A new algorithm for property checking. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 117–127. Springer, Heidelberg (2006)
14. Heizmann, M., Hoenicke, J., Podelski, A.: Nested interpolants. In: POPL (2010)
15. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy abstraction. In: POPL (2002)
16. Kidd, N., Lal, A., Reps, T.: WALi: The Weighted Automaton Library (2007), http://www.cs.wisc.edu/wpis/wpds/download.php
17. Kruegel, C., Kirda, E., Mutz, D., Robertson, W., Vigna, G.: Automating mimicry attacks using static binary analysis. In: USENIX Sec. Symp. (2005)
18. Lim, J., Lal, A., Reps, T.: Symbolic analysis via semantic reinterpretation. In: SPIN Workshop (2009)
19. Lim, J., Reps, T.: A system for generating static analyzers for machine instructions. In: Hendren, L. (ed.) CC 2008. LNCS, vol. 4959, pp. 36–52. Springer, Heidelberg (2008)
20. Linn, C., Debray, S.: Obfuscation of executable code to improve resistance to static disassembly. In: CCS (2003)
21. Müller-Olm, M., Seidl, H.: Analysis of modular arithmetic. In: Sagiv, M. (ed.) ESOP 2005. LNCS, vol. 3444, pp. 46–60. Springer, Heidelberg (2005)
22. Ramalingam, G., Field, J., Tip, F.: Aggregate structure identification and its application to program analysis. In: POPL (1999)
23. Reps, T., Balakrishnan, G., Lim, J.: Intermediate-representation recovery from low-level code. In: PEPM (2006)
24. Srivastava, A., Edwards, A., Vo, H.: Vulcan: Binary transformation in a distributed environment. MSR-TR-2001-50, Microsoft Research (April 2001)
25. Thakur, A., Lim, J., Lal, A., Burton, A., Driscoll, E., Elder, M., Andersen, T., Reps, T.: Directed proof generation for machine code. TR 1669, UW-Madison (April 2010)

Verifying Low-Level Implementations of High-Level Datatypes

Christopher L. Conway and Clark Barrett

New York University, Dept. of Computer Science
{cconway,barrett}@cs.nyu.edu

Abstract. For efficiency and portability, network packet processing code is typically written in low-level languages and makes use of bit-level operations to compactly represent data. Although packet data is highly structured, low-level implementation details make it difficult to verify that the behavior of the code is consistent with high-level data invariants. We introduce a new approach to the verification problem, using a high-level definition of packet types as part of a specification rather than an implementation. The types are not used to check the code directly; rather, the types introduce functions and predicates that can be used to assert the consistency of code with programmer-defined data assertions. We describe an encoding of these types and functions using the theories of inductive datatypes, bit vectors, and arrays in the CVC3 SMT solver. We present a case study in which the method is applied to open-source networking code and verified within the CASCADE verification platform.

1 Introduction

Packet-level networking code is critical to communications infrastructure and vulnerable to malicious attacks. This code is typically written in low-level languages like C or C++. Packet fields are "parsed" using pointer arithmetic and bit-wise operators to select individual bytes and sequences of bits within a larger untyped buffer (e.g., a `char` array). This approach yields high-performance, portable code, but can lead to subtle errors.

An alternative is to write packet-processing code in special-purpose high-level languages, e.g., binpac [17], Melange [13], Morpheus [1], or Prolac [9]. These languages typically provide a facility for describing network packets as a set of nested, and possibly recursive, datatypes. The language compilers then produce low-level packet-processing code which aims to match or exceed the performance of the equivalent hand-coded C/C++. This requires an expensive commitment to rewriting existing code.

We propose a new approach, one which fuses the power of higher-level datatypes with the convenience and efficiency of legacy code. The key idea is to use a high-level description of "packet types" as the basis for a *specification*, not an *implementation*. Instead of using a compiler to try to reproduce a performant implementation, we can annotate the existing implementation to indicate the intended high-level semantics, then verify that the implementation is consistent

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 306–320, 2010.

with those semantics. We make use of the theories of inductive datatypes, bit vectors, and arrays in CVC3 to encode the relationship between the high-level and low-level semantics. Using this encoding, it is possible to verify that the low-level code represents, in essence, an implementation of a well-typed high-level specification.

In this paper we will present our proposed notation for defining packet datatypes and stating datatype invariants in C code. We describe the translation of the datatype definition and code assertions into verification conditions in the CVC3 SMT solver. The encoding relies crucially on automatically generated separation invariants, which allow CVC3 to efficiently reason about recursive data structures without producing false assertion failures due to spurious aliasing relationships. Finally, we present a case study applying our approach to real code from the BIND DNS server. We are able to verify high-level data invariants of the code with reasonable efficiency. To our knowledge, no other verification tool is capable of automatically proving such datatype invariants on existing C code.

2 A Motivating Example

Figure 1(a) illustrates the definition of a simple, high-level list datatype in a notation similar to that of languages like ML and Haskell. The type has two constructors: `cons`, which creates a list node with an associated `data` array and a `cdr` field representing the remainder of the list, and `nil`, which represents an empty list. Figure 1(b) gives the high-level pseudo-code for a function that computes the length of a list, defined as the number of `cons` values encountered via `cdr` "links" before a `nil`. The code simply checks whether `lst` is a `cons` value using the "tester" function `isCons`. If it is, it increments the length and updates `lst` using the `cdr` field. If it is not, it returns the computed length.

In a high-level language, the compiler is given the freedom to implement datatypes like `List` as it chooses, typically using linked heap structures to represent individual datatype values. The programmer concentrates on the high-level semantics of the algorithm, allowing the compiler to encode and decode the data. By contrast, in packet processing code, the datatype is defined in terms of an explicit data layout. The data is "packed" into a contiguously allocated block of memory. The high-level algorithm and the encoding and decoding of data are intertwined.

The `List` type in Fig. 1(c) illustrates a simple "packed" linked list implementation. Like the definition in Fig. 1(a), `List` is a union type with two variants. However, instead of simply declaring a set of data fields, each variant explicitly defines its own representation. The representation of a `cons` value is: a 1-bit `tag` field (the highest-order bit of the first byte), a 7-bit `count` field (the lower-order bits of the first byte), a `data` field of exactly `count` bytes, and another `List` value `cdr`, which follows immediately in memory. The value of `tag` is constrained by the constant bit vector value `0b1`. The constraint requires the `tag` bit of a `cons` value always to be `1`. The representation of a `nil` value has a similar constraint: a `nil` value consists of a single 8-bit `tag` field, which must be `0x00`. The fact

```
type List =
  cons {
    count: Nat,
    data: Int array,
    cdr: List
  }
| nil
```

(a)

```
Nat list_length(List lst) {
    Nat count = 0;
    while( isCons(lst) ) {
        count++;
        lst = cdr(lst);
    }
    return count;
}
```

(b)

```
type List =
  cons {
    tag:1 = 0b1,
    count: 7,
    data: u_char[count],
    cdr: List
  }
| nil {
    tag:8 = 0x00
  }
```

(c)

```
u_int list_length(const u_char *p) {
    u_int n, count = 0;
    while( (n = *p++) & 0x80 ) {
        { isCons(prev(p)) }
        count++;
        p += n & 0x7f;
        { toList(p) = cdr(prev(p)) }
    }
    if( n != 0 ) // malformed list
        return (-1);
    { isNil(p) }
    return count;
}
```

(d)

Fig. 1. Defining and using a simple linked list datatype

that the `tag` bit of a `cons` value must be `1` while the bits of a `nil` value must all be `0` ensures that we can unambiguously decode `cons` and `nil` values. (A full grammar for "packed" datatype definitions is given in Section 3.1.)

Figure 2 illustrates the interpretation of a sequence of bytes as a `List` value. The first byte (`0x82`) has its high bit set; thus, it is a `cons` value. The low-order bits tell us that `count` is `2`; thus, `data` has two elements: `0x01` and `0x02`. The `cdr` field is another `List` value, encoded starting at the next byte. This byte (`0x81`) is also a `cons` value, since it also has its high bit set. Its `count` field is `1`, its `data` field the single element `0x03`. Its `cdr` is the `List` value at the next byte (`0x00`), a `nil` value.

Figure 1(d) gives a low-level implementation of the length function, which operates over the implicit `List` value pointed to by the input `p`. (The bracketed, italicized portions of the code are verification annotations, which are described in Section 3.3.) Note that the structure of the function is very similar to the code in Fig. 1(b), but that high-level operations have been replaced by their low-level equivalents—pointer arithmetic and bit-masking operations are used to detect constructors and select fields. A notable addition is the `if` statement that appears after the `while` loop. In the high-level code, we could assume that the data was well-formed, i.e., that every list is either a `cons` or a `nil` value. In the low-level implementation, we may encounter byte sequences which are not assigned a meaning by the datatype definition—in this case a non-zero byte in which the high bit is not set, which satisfies the data constraints of neither `cons` nor `nil`. The function handles this erroneous case by returning an error code.

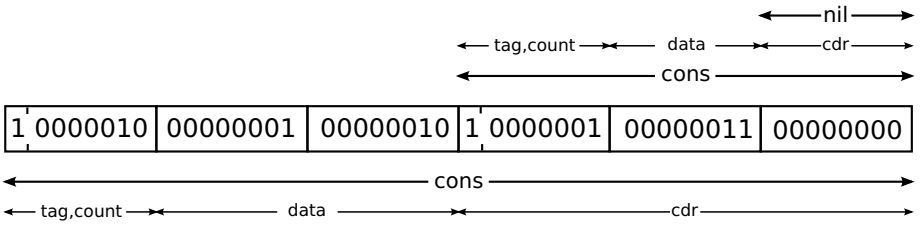


Fig. 2. The layout of a `List` value

The challenge, in essence, is to prove that the low-level code in Fig. 1(d) is a refinement of the high-level code in Fig. 1(b). To this end, we need to build a bridge between the high-level semantics of the datatype and the low-level implementation.

3 Our Approach

The verification process proceeds in four steps:

1. The programmer provides a datatype declaration, as in Fig. 1(c), defining the high-level structure and layout of the data.
2. Using the datatype declaration, we generate a set of CVC3 declarations and axioms encoding the relationship between the high-level type and its implementation.
3. The programmer adds code annotations specifying the expected behavior of the low-level code, in terms of functions derived from the datatype definition.
4. We use the CASCADE verification platform to translate the code and annotations into a set of verification conditions to be checked by CVC3. If all of the verification conditions are valid, then the code satisfies the specification.

3.1 Datatype Definition

Figure 3 gives the full grammar for datatype definitions. The notation for datatype definitions is similar to that of disjoint union types in higher-level languages like ML and Haskell. There is an important distinction: unlike datatype implementations generated by compilers, it is up to the user to ensure that the encoding of values is unambiguous and consistent. The declaration should provide all of the information needed both to encode a datatype value as a sequence of bytes and to decode a well-formed sequence of bytes as a high-level datatype value.

A type consists of a set of constructors. Each constructor has a set of fields. A field type is one of four kinds: a bit vector of constant integer size, a plain C scalar type, an array of C type elements, or another datatype. (The syntax of C type declarators is that of ANSI/ISO C [2].) Bit vectors and C types may have value constraints. Bit vector constants are preceded by `0b` (for binary constants) or `0x` (for hexadecimal constants). Arrays have a length: either a constant integer or the value of a prior field—the declaration language supports a limited form of dependent types.

$$
\begin{aligned}
\mathit{Type} &::= \texttt{type}\ \mathit{Id}\ \texttt{=}\ \mathit{Cons}\ (\texttt{|}\ \ \mathit{Cons})* \\
\mathit{Cons} &::= \mathit{Id}\ \texttt{\{}\ \ \mathit{Field}\ (\texttt{,}\ \mathit{Field})*\ \texttt{\}} \\
\mathit{Field} &::= \mathit{Id}\ \texttt{:}\ \mathit{FieldType} \\
\mathit{FieldType} &::= \mathit{BvType} \mid \mathit{CType} \mid \mathit{ArrType} \mid \mathit{TypeId} \\
\mathit{BvType} &::= \mathit{IntConst}\ (\texttt{=}\ \mathit{BvConst})? \\
\mathit{BvConst} &::= \texttt{0b[01]+} \mid \texttt{0x[0-9a-fA-F]+} \\
\mathit{CType} &::= \mathit{CScalarType}(\texttt{=}\ \mathit{CConst})? \\
\mathit{ArrType} &::= \mathit{CType}\ \texttt{[}\mathit{ArrLength}\texttt{]} \\
\mathit{ArrLength} &::= \mathit{IntConst} \mid \mathit{Id} \\
\mathit{TypeId} &::= \mathit{Id}
\end{aligned}
$$

Fig. 3. Grammar for datatype definitions

$$
\begin{array}{l}
\textbf{datatype}\ \ \mathit{List} = \ \mathit{cons}\ \{\ \mathit{count} : \mathcal{BV}_7,\ \ \mathit{data} : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array},\ \ \mathit{cdr} : \mathit{List}\ \} \\
\qquad\qquad\qquad\quad \mid\ \mathit{nil} \\
\qquad\qquad\qquad\quad \mid\ \mathit{undefined}
\end{array}
$$

$$
\begin{array}{rlrl}
\mathit{toList} : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array} \times \mathcal{BV}_N \to \mathit{List} & & m : (\mathcal{BV}_N, \mathcal{BV}_8)\ \mathsf{array} \\
\mathit{sizeOfList} : \mathit{List} \to \mathbb{N} & & \ell : \mathcal{BV}_N
\end{array}
$$

$$
\begin{array}{rcll}
\multicolumn{4}{l}{\textbf{let}\ x = \mathit{toList}(m, \ell)\ \textbf{in}} \\
\mathit{isCons}(x) & \iff & m[\ell][7] & (\textsc{ConsTest}) \\
\mathit{isNil}(x) & \iff & m[\ell] = 0 & (\textsc{NilTest}) \\
\mathit{isCons}(x) & \implies & \mathit{count}(x) = m[\ell][6{:}0] & (\textsc{ConsSel}) \\
 & & \quad \wedge\ (\forall 0 \le i < \mathit{count}(x).\ \ \mathit{data}(x)[i] = m[\ell + i + 1]) & \\
 & & \quad \wedge\ \ \mathit{cdr}(x) = \mathit{toList}(m, \ell + \mathit{count}(x) + 1) &
\end{array}
$$

$$
\begin{aligned}
\mathit{sizeOfList}(\mathit{cons}(\mathit{count}, \_, \mathit{cdr})) &= 1 + \mathit{count} + \mathit{sizeOfList}(\mathit{cdr}) && (\textsc{ConsSize}) \\
\mathit{sizeOfList}(\mathit{nil}) &= 1 && (\textsc{NilSize}) \\
\mathit{sizeOfList}(\mathit{undefined}) &= 0 && (\textsc{UndefSize})
\end{aligned}
$$

Fig. 4. Datatype definition and axioms for the type `List`

3.2 Translation to Cvc3

It is straightforward to translate the datatype definition into an inductive datatype in the input language of Cvc3. The translation for the `List` datatype is given in Fig. 4. We use $\mathbb{N}$ to denote the type of natural numbers; $\mathcal{BV}_k$ to denote the type of bit vectors of size k (i.e., k-tuples of booleans); and $(\alpha, \beta)\ \mathsf{array}$ to denote the type of arrays with indices of type α and elements of type β. We use N to denote the (platform-dependent) size of a pointer (i.e., the type of pointers is $\mathcal{BV}_N$). For an array a, $a[i]$ denotes the element of a at index i; similarly, for a bit vector b, $b[i]$ denotes the ith bit of b and $b[j{:}i]$ denotes the *extraction* of bits i through j (the result is a bit vector of size $j - i + 1$). The size of the result of arithmetic operations on bit vectors is the size of the larger operand; the smaller operand is implicitly zero-extended. When used in an integer context, bit vectors are interpreted as unsigned.

The translation produces a Cvc3 datatype definition reflecting the data layout of the declaration augmented with an explicit *undefined* value. Note that the `tag`

fields are omitted from the definition—since they are constrained by constants, they are only needed to decode the high-level data value.

CVC3 automatically generates a set of datatype testers and field selectors. The testers *isCons*, *isNil*, and *isUndefined* are predicates that hold for a *List* value x iff x is, respectively, a *cons*, *nil*, or *undefined* value. The selectors *count*, *data*, and *cdr* are functions that map a *List* value to the value of corresponding field.

Note that the definition of *List* itself does not include any data constraints on field values. These constraints are introduced by the function *toList*, which maps a pointer-indexed array of bytes m and a location ℓ to the *List* value represented by the sequence of bytes starting at ℓ in m. The axioms ConsTest and NilTest enforce the data constraints on the `tag` fields of `cons` and `nil`, respectively. The axiom ConsSel represents the encoding of the remaining fields of `cons`. Note that there is no explicit rule for the value *undefined*: if the data constraints given in ConsTest and NilTest do not apply, then the only remaining value that *toList* can return is *undefined*.

The function *sizeOfList* maps a *List* value to the size of its encoding in bytes. By convention, the size of *undefined* is 0.

3.3 Code Assertions

The functions generated by the CVC3 translation are exposed in the assertion language as functions that take a single pointer argument. In the case of the function `toList`, the additional array argument, representing the configuration of memory, is introduced in the verification condition translation. The pointer argument of the other functions is implicitly converted to a `List` value using `toList`. The assertion language also provides auxiliary functions `init` and `prev`, mapping variables to their initial values in, respectively, the current function and loop iteration.

Returning to the code in Fig. 1(d), the bracketed, italicized assertions state the expected high-level semantics of the implementation. Specifically, they assert:

- The loop test succeeds only for `cons` values.
- The body of the loop sets `p` to the `cdr` of its initial value in each loop iteration.
- If the value is well-formed, then `p` points to a `nil` value when the function returns.

The functions representing testers rely on the data constraints of the type, e.g., `p` points to a `cons` value iff the byte sequence pointed to by `p` satisfies the data constraints of `cons` (i.e., the high bit of `*p` is set). The functions representing testers rely on the structure of the type, e.g., `toList(q)==cdr(p)` iff `p` points to a `cons` value and `q==p+count(p)+1`.

Loops can be annotated with invariants: we can separately prove initialization and preservation of the invariant, and that each assertion in the body of the loop is valid when the invariant is assumed on entry.

3.4 Verification Condition Generation

The final verification step is to use the CASCADE verification platform to translate the code and assertions into formulas that can be validated by CVC3. Verification is driven by a *control file*, which defines a set of paths to check and allows annotations and assertions to be injected at arbitrary points along a path. Each code assertion is transformed into a verification condition, which is passed to CVC3 and checked for validity. For each condition, CVC3 will return "valid" (the condition is always true), "invalid" (the condition is not always true), or "unknown" (due to incompleteness, CVC3 could not prove invalidity). CASCADE returns "valid" for a path iff CVC3 returns "valid" for every assertion on the path. If CVC3 returns "invalid" or "unknown" for any assertion, CASCADE returns "invalid", along with a counterexample.

Note 1. Since the background axioms that define datatypes are universally quantified, deciding validity of the generated verification conditions is undecidable in general. CVC3 will never return "invalid" for any verification condition that it cannot prove valid; instead, it will return "unknown" when a pre-determined instantiation limit is reached. There are fragments of first-order logic that are decidable with instantiation-based algorithms [6]. Encoding the datatype assertions in a decidable fragment of first-order logic is a subject for future work.

CASCADE supports a number of encodings for C expressions and program semantics. For datatype verification, we make use of a bit vector encoding, which is parameterized by the platform-specific size of a pointer and of a memory word.

An additional consideration is the memory model used in the verification condition. The memory model specifies the interpretation of pointer values and the effect of memory accesses (both reads and writes) on the program state. A memory model may abstract away details of the program's concrete semantics (e.g., by discarding information about the precise layout of structures in memory) or it may refine the concrete semantics (e.g., by choosing a deterministic allocation strategy). We discuss the memory model in detail in the next section.

4 Memory Model

In order to accurately reflect the datatype representation, we require a memory model that is bit-precise. At the same time, to avoid a blow-up in verification complexity and overly conservative results, we would like a relatively high-level model that preserves the separation invariants of the implementation. To this end, we define a memory model based on separation analysis [7] that we call a *partitioned heap*.

The flat model. First, we will define for comparison a simple model which is self-evidently sound. A *flat memory model* interprets every pointer expression as a bit vector of size N. Every allocated object in the program is associated with a region of memory (i.e., a contiguous block of locations) distinct from

all previously allocated regions. The state of memory is modeled by a single pointer-indexed array m. The value stored at location ℓ is thus $m[\ell]$.

Using the flat memory model, we can translate the first assertion in Fig. 1(d) into the verification condition

$$m_1 = m_0[\texttt{\&p} \mapsto m_0[\texttt{\&p}] + 1] \wedge m_2 = m_1[\texttt{\&n} \mapsto m_0[m_0[\texttt{\&p}]]] \wedge m_2[\texttt{\&n}][7] \\ \implies \mathit{isCons}(\mathit{toList}(m_2, m_0[\texttt{\&p}]))$$

where we use `&x` to denote the location in memory of the variable `x` (i.e., its *lvalue*) and $a[i \mapsto e]$ to denote the update of array a with element e at index i. Assuming `&p`, `&n`, and m[`&p`] are distinct, the validity of the formula is a direct consequence of the axiom ConsTest.

The flat model accurately represents unsafe operations like casts between incompatible types and bit-level operations on pointers. However, it is a very weak model—its lack of guaranteed separation between objects makes it difficult to prove strong properties of data-manipulating programs.

Example 1. Consider the Hoare triple

$$\{\ \texttt{toList(q)==cdr(p)}\ \}\ \texttt{i++}\ \{\ \texttt{toList(q)==cdr(p)}\ \}$$

where `p` and `q` are known to not alias `i`. In a flat memory model, this is interpreted as

$$\mathit{toList}(m_0, m_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_0, m_0[\texttt{\&p}])) \\ \wedge\ m_1 = m_0[\texttt{\&i} \mapsto m_0[\texttt{\&i}] + 1] \\ \implies \mathit{toList}(m_1, m_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(m_1, m_1[\texttt{\&p}]))$$

Since *toList* is defined axiomatically using recursion (see Fig. 4), it is not immediately obvious that the necessary lemma

$$\mathit{toList}(m_0, m_0[\texttt{\&p}]) = \mathit{toList}(m_1, m_1[\texttt{\&p}])$$

is implied (similarly for `q`). Even if `p` and `q` can never point to `i`, we cannot rule out the possibility that the `List` values pointed to by `p` and `q` depend in some way on the value of `i`. Now, suppose we add the assumption

`allocated(p,p+sizeOfList(p))`,

where `allocated(x,y)` means that pointer `x` is the base of a region of memory, disjoint from all other allocated regions, bounded by pointer `y`. Even then, the proof of the assertion relies on the following theorem, which is beyond the capability of automated theorem provers like CVC3 to prove:

$$(\forall y : x \le y \le x + \mathit{sizeOfList}(\mathit{toList}(m_0, x)) : m_0[y] = m_1[y]) \\ \implies \mathit{toList}(m_0, x) = \mathit{toList}(m_1, x)$$

□

What we require is a separation invariant allowing us to apply the "frame rule" of separation logic [19,15]:

$$\{\ \texttt{toList(q)==cdr(p)*i==}v\ \}\ \texttt{i++}\ \{\ \texttt{toList(q)==cdr(p)*i==}v+1\ \}$$

where `*` denotes *separating conjunction*: $A{*}B$ holds iff memory can be partitioned into two disjoint regions R and R' where A and B hold, respectively.

The partitioned model. The separation invariants we need can be obtained using separation analysis [7]. The analysis can be understood as the inverse of *may-alias analysis* [10,11]: if pointers `p` and `q` can never alias, then the objects they point to must be separated (i.e., they occupy disjoint regions of memory).

The output of the separation analysis is a *partition* $P = \{P_1, \ldots, P_k\}$, where each P_i represents a disjoint region of memory, and a map from pointer expressions to regions—if expression E is mapped to partition P_i, then E can only point to objects allocated in region P_i. If the separation analysis maps pointers expressions E and E' to different partitions, then E and E' cannot be aliased in any well-defined execution of the program.

A *P-partitioned memory model* for partition $P = \{P_1, \ldots, P_k\}$ interprets every pointer expression as a pair $(\ell, i) \in \mathcal{BV}_N \times \mathbb{N}$, where ℓ is a location and i is a partition index. The state of memory is modeled by a collection of pointer-indexed arrays $\langle m_1, \ldots, m_k \rangle$. The location pointed to by pointer expression (ℓ, i) is the array element $m_i[\ell]$.

Example 2. The program in Fig. 1(d) can be divided into two partitions. The first partition contains the parameter `p` and local variables `n` and `size`. The second partition contains the object pointed-to by `p`. We represent the two partitions by two memory arrays, s and h, respectively. Thus, the value of the variable `n` is represented by the array element $s[\texttt{\&n}]$; the value of the expression `*p` is represented by the array element $h[s[\texttt{\&p}]]$.

A partitioned memory model solves the problem of Example 1 by isolating the *List* value in its own partition:

$$\begin{aligned} \mathit{toList}(h_0, s_0[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_0[\texttt{\&p}])) \wedge s_1 = s_0[\texttt{\&i} \mapsto s_0[\texttt{\&i}] + 1] \\ \implies \mathit{toList}(h_0, s_1[\texttt{\&q}]) = \mathit{cdr}(\mathit{toList}(h_0, s_1[\texttt{\&p}])) \end{aligned}$$

Given that `&p`, `&q` and `&i` are distinct, the formula is trivially valid. □

We say a program is *memory safe* if all reads and writes through pointers occur only within allocated objects. Like pointer analysis, the soundness of the separation analysis is conditional on memory safety. Thus, the soundness of verification using a partitioned memory model will likewise be conditional on memory safety.

It may seem questionable to attempt to verify a program using information which depends for its correctness on prior verification of the same program. In previous work, we showed that a sound combination is possible, as long as the verification procedure ensures that no memory safety errors occur along the path under consideration [5]. It is thus essential that the verification conditions

include assertions that establish the memory safety of the statements along each path in the program.

In our experience, a partitioned memory model can make an order-of-magnitude difference in verification time compared to a flat memory model—indeed, properties are provable by CVC3 using a partitioned model that cannot be proved using a flat model (see Section 5.1).

5 Case Study: Compressed Domain Names

To demonstrate the utility of our approach, we will describe a more complex application, taken from real code. We will show the definition of a real-world datatype, the annotations for a function operating on that datatype, and the results of using CASCADE to verify the function.

A definition for the datatype `Dn`, representing an RFC 1035 *compressed domain name* [14], is given in Fig. 5. `Dn` is a union type with three variants: `label`, `indirect`, and `nullt`. The representation of a `label` value is: a 2-bit `tag` field (which must be zeroes), a 6-bit `len` field (which must *not* be all zeroes), a `label` field of exactly `len` bytes, and another `Dn` value `rest`, which follows immediately in memory. An `indirect` value has a 2-bit `tag` (which must be `0b11`) and a 14-bit offset. A `nullt` value has only an 8-bit `tag`, which must be zero. The constraints on the `tag` fields of `label`, `indirect`, and `nullt` allow us to distinguish between values.

```
type Dn =
    label {
      tag:2 = 0b00,
      len:6 != 0b000000,
      name:u_char[len],
      rest:Dn
    }
  | indirect {
      tag:2 = 0b11,
      offset:14
    }
  | nullt {
      tag:8 = 0x00
    }
```

Fig. 5. Definition of the `Dn` datatype

Consider the function `ns_name_skip` in Fig. 6. The low-level pointer and bit-masking operations represent the traversal of the high-level `Dn` data structure. The correctness of the implementation is properly expressed in terms of that data structure.

In terms of the type `Dn`, the code in Fig. 6 is straightforward. The pointer `cp`, the value pointed to by the parameter `ptrptr`, points to a `Dn` value. The loop test (Line 12) assigns the first byte of the value to the variable `n` and advances `cp` by one byte. If `n` is `0`, then `cp` pointed to a `nullt` value and the loop exits. Otherwise (Line 14), the `switch` statement checks the two most significant bits of `n`—the `tag` field of a `label` or `indirect` value. If the `tag` field contains zeroes

```
1  #define NS_CMPRSFLAGS (0xc0)
2
3  int
4  ns_name_skip(const u_char **ptrptr, const u_char *eom) {
5      { allocated(*ptrptr, eom) }
6      const u_char *cp;
7      u_int n;
8
9      cp = *ptrptr;
10     { @invariant: cp ≤ eom ⟹
11         cp + sizeOfDn(cp) = init(cp) + sizeOfDn(init(cp)) }
12     while (cp < eom && (n = *cp++) != 0) {
13         /* Check for indirection. */
14         switch (n & NS_CMPRSFLGS) {
15           case 0:               /* normal case, n == len */
16             { isLabel(prev(cp)) }
17             cp += n;
18             { rest(prev(cp)) = toDn(cp) }
19             continue;
20         case NS_CMPRSFLGS:      /* indirection */
21             { isIndirect(prev(cp)) }
22             cp++;
23             break;
24         default:                /* illegal type */
25             __set_errno (EMSGSIZE);
26             return (-1);
27         }
28         break;
29     }
30     if (cp > eom) {
31         __set_errno (EMSGSIZE);
32         return (-1);
33     }
34     { cp = eom ∨ cp = init(cp) + sizeOfDn(init(cp)) }
35     *ptrptr = cp;
36     return (0);
37 }
```

Fig. 6. The function `ns_name_skip` from BIND

(Line 15), `cp` is advanced past the `label` field to point to the `Dn` value of the `rest` field. If the `tag` field contains ones (Line 20), `cp` is advanced past the `offset` field and breaks the loop. The `default` case of the `switch` statement returns an error code—the `tag` field was malformed. At the end of the loop, if `cp` has not exceeded the bound `eom`, the value of `cp` is one greater than the address of the last byte of the `Dn` value that `cp` pointed to initially. This is the contract of the function: given a reference to a pointer to a valid `Dn` value, it advances the pointer past the `Dn` value or to the bound `eom`, whichever comes first, and returns `0`; if the `Dn` value is invalid, it returns `-1`.

Annotating the source code. The datatype definition is translated into an inductive datatype with supporting functions and axioms, as in Section 3.2. The translation generates testers *isLabel*, *isIndirect*, and *isNullt*; selectors *len*, *name*, *rest*, etc.; and the encoding functions *toDn* and *sizeOfDn*. Each of these functions is now available for use in source code assertions, as in the bracketed, italicized portions in Fig. 6.

The annotations in Fig. 6 also make reference to some auxiliary functions: `init(x)` represents the initial value of a variable `x` in the function; `prev(x)`

$$
\begin{array}{rl}
& s_0[\texttt{\&cp}] \geq s_0[\texttt{\&eom}] \\
\vee\ & s_0[\texttt{\&cp}] + sizeOfDn(toDn(h_0, s_0[\texttt{\&cp}])) \\
= & init(\texttt{\&cp}) + sizeOfDn(toDn(h_0, init(\texttt{\&cp}))) \qquad (1) \\
& s_0[\texttt{\&cp}] < s_0[\texttt{\&eom}] \qquad (2) \\
& s_1 = s_0[\texttt{\&cp} \mapsto s_0[\texttt{\&cp}] + 1] \qquad (3) \\
& s_2 = s_1[\texttt{\&n} \mapsto h_0[s_0[\texttt{\&cp}]]] \qquad (4) \\
& s_2[\texttt{\&n}] \neq 0 \qquad (5) \\
& s_2[\texttt{\&n}][7:6] = 0 \qquad (6) \\
& is\_label(toDn(h_0, s_0[\texttt{\&cp}])) \qquad (7) \\
& s_3 = s_2[\texttt{\&cp} \mapsto s_2[\texttt{\&cp}] + s_2[\texttt{\&n}]] \qquad (8) \\
& rest(toDn(h_0, s_0[\texttt{\&cp}])) = toDn(h_0, s_3[\texttt{\&cp}]) \qquad (9) \\
\hline
& s_4[\texttt{\&cp}] \geq s_4[\texttt{\&eom}] \\
\vee\ & s_4[\texttt{\&cp}] + sizeOfDn(toDn(h_0, s_4[\texttt{\&cp}])) \\
= & init(\texttt{\&cp}) + sizeOfDn(toDn(h_0, init(\texttt{\&cp}))) \qquad (10)
\end{array}
$$

Fig. 7. Verification conditions for `ns_name_skip`

represents the previous value of a variable `x` in a loop (i.e., the value at the beginning of an iteration).

On entry to the function (Line 5), we assume that the region pointed to by `*ptrptr` and bounded by `eom` is properly allocated. To each `switch` case (Lines 15 and 20), we add an assertion stating that the observed `tag` value (i.e, `n & NS_CMPRSFLGS`) is consistent with a particular datatype constructor (i.e., `label` or `indirect`). (Note that `prev(cp)` refers to the value of `cp` *before* the loop test, which has side effects). The loop invariant (Lines 10-11) states that `cp` advances through the `Dn` data structure pointed to by `init(cp)`—in each iteration of the loop, if `cp` has not exceeded the bound `eom`, it points to a `Dn` structure (perhaps the "tail" of a larger, inductive value) that is co-terminal with the structure pointed to by `init(cp)`. On termination, the loop invariant implies the desired post-condition: if no error condition has occurred, `*ptrptr` will point to the byte immediately following the `Dn` value pointed to by `init(cp)`—the pointer will have "skipped" the value. Note that we do not require an assertion stating that `cp` is reachable from `init(cp)` via `rest` "pointers" to prove the desired property—the property is provable using purely inductive reasoning.

Using the code annotations, Cascade can verify the function by generating a set of verification conditions representing non-looping static paths through the function. Fig. 7 gives an example of such a verification condition. It represents the path from the head of the loop through the `0` case of the `switch` statement (Line 15), ending with the `continue` statement and asserting the preservation of the loop invariant. (Note that we assume here that pointers are 8 bits. Larger pointer values are easily handled, but the formulas are more complicated.) As in Section 4, the verification condition uses a partitioned memory model with two memory arrays, s and h: the values of local variables and parameters are stored in s while the `Dn` value pointed to by `cp` is stored in h. Proposition (1) asserts the loop invariant on entry. Propositions (2)–(5) represent the evaluation, including effects, of the loop test. Proposition (6) represents the matching of the `switch` case. Propositions (7)–(9) capture the body of the case block. Finally,

Proposition (10) (the proposition we would like to prove, given the previous assumptions) asserts the preservation of the loop invariant.

5.1 Experiments

Table 1 shows the time taken by CVC3 to prove the verification conditions generated by CASCADE for `ns_name_skip`, using both the flat and partitioned memory models. The times given are for a Intel Dual Core laptop running at 2.2GHz with 4GB RAM. Each VC represents a non-looping, non-erroneous path to an assertion. The two TERM VCs represent the loop exit paths: TERM (1) is the path where the first conjunct is false (`cp >= eom`; TERM (2) is the path where the first conjunct is true (`cp < eom`) and the second is false (`n == 0`). The verification conditions marked with * for the flat memory model timed out after two minutes—we believe that these formulas are not provable in CVC3. All of the verification conditions together can be validated using the partitioned memory model in less than one second.

Table 1. Running times on `ns_name_skip` VCs

Name	Lines	Time (seconds) Flat	Part.
INIT	5–12	0.34	0.03
CASE `0` (1)	12-16	13.94	0.05
CASE `0` (2)	12-28	33.42	0.06
CASE `0` (3)	12-19	*	0.12
CASE `0xc0` (1)	12–14, 20–21	6.14	0.04
CASE `0xc0` (2)	12–14, 20–23, 30, 34	*	0.07
TERM (1)	12, 30, 34	0.63	0.06
TERM (2)	12, 30, 34	*	0.05

6 Related Work

Some early work on verification of programs operating on complex datatypes was done by Burstall [4], Laventhal [12], and Oppen and Cook [16]. Their work assumes that data layout is an implementation detail that can be abstracted away. Our work here focuses on network packet processing code, where the linear layout of the data structure is an essential property of the implementation.

More recently, O'Hearn, Reynolds, and Yang [15] have approached the problem using *separation logic* [19,8]. Given assumptions about the structure of the heap, the logic allows for powerful localized reasoning. In this work, we use separation analysis in the style of Hubert and Marché [7] to establish separation invariants, thus "localizing" the verification conditions.

Rakamaric and Hu [18] describe a variation of Burstall's memory model [4,3] suitable for bit-precise verification of low-level code. However, their approach relies on a compile-time type analysis of the program—since we are trying to

verify datatypes that are not explicitly represented in the program code, we must rely on a more primitive model.

7 Conclusions

In this paper, we have presented a novel approach to the verification of low-level packet processing code. Instead of rewriting code in a high-level declarative language, we propose to apply the information derived from a declarative specification to enable checking high-level assertions embedded in the low-level implementation. The approach allows for the continued use of tested, performant code, with the increased assurance of verification. The experimental results are encouraging; we believe our technique can scale to several hundreds or thousands of lines of code. In future work, we intend to extend our technique to a broader class of datatypes, including more typical pointer-linked data structures.

Acknowledgments

This work was supported in part by the National Science Foundation under Grant No. 0644299. The authors would like to thank Dejan Jovanović for his help debugging the CVC3 translation and preparing the figures for this paper.

References

1. Abbott, M.B., Peterson, L.L.: A language-based approach to protocol implementation. IEEE/ACM Trans. Netw. 1(1), 4–19 (1993)
2. American National Standard for Programming Languages - C, ANSI/ISO 9899-1990 (August 1989)
3. Bornat, R.: Proving pointer programs in Hoare logic. In: Mathematics of Program Construction, pp. 102–126 (2000)
4. Burstall, R.: Some techniques for proving correctness of programs which alter data structures. In: Machine Intelligence (1972)
5. Conway, C.L., Dams, D., Namjoshi, K.S., Barrett, C.: Points-to analysis, conditional soundness, and proving the absence of errors. In: Alpuente, M., Vidal, G. (eds.) SAS 2008. LNCS, vol. 5079, pp. 62–77. Springer, Heidelberg (2008)
6. Ge, Y., de Moura, L.: Complete instantiation for quantified formulas in Satisfiability Modulo Theories. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 306–320. Springer, Heidelberg (2009)
7. Hubert, T., Marché, C.: Separation analysis for deductive verification. In: HAV, March 2007, pp. 81–93 (2007)
8. Ishtiaq, S.S., O'Hearn, P.W.: Bi as an assertion language for mutable data structures. SIGPLAN Not. 36(3), 14–26 (2001)
9. Kohler, E., Kaashoek, M.F., Montgomery, D.R.: A readable TCP in the Prolac protocol language. Computer Communication Review 29(4), 3–13 (1999)
10. Landi, W., Ryder, B.G.: Pointer-induced aliasing: a problem taxonomy. In: POPL, January 1991, pp. 93–103 (1991)

11. Landi, W., Ryder, B.G.: A safe approximate algorithm for interprocedural aliasing. In: Programming Language Design and Implementation (PLDI), June 1992, pp. 235–248 (1992)
12. Laventhal, M.S.: Verifying programs which operate on data structures. In: Reliable Software, pp. 420–426 (1975)
13. Madhavapeddy, A., Ho, A., Deegan, T., Scott, D., Sohan, R.: Melange: creating a "functional" internet. In: European Conf. on Comp. Sys. (EuroSys), pp. 101–114 (2007)
14. Mockapetris, P.: Domain names - implementation and specification. RFC 1035 (Standard) (November 1987)
15. O'Hearn, P.W., Reynolds, J.C., Yang, H.: Local reasoning about programs that alter data structures. In: Fribourg, L. (ed.) CSL 2001. LNCS, vol. 2142, pp. 1–19. Springer, Heidelberg (2001)
16. Oppen, D.C., Cook, S.A.: Proving assertions about programs that manipulate data structures. In: Symposium on the Theory of Computing (STOC), pp. 107–116 (1975)
17. Pang, R., Paxson, V., Sommer, R., Peterson, L.: binpac: a yacc for writing application protocol parsers. In: Internet Measurement Conf. (IMC), pp. 289–300 (2006)
18. Rakamaric, Z., Hu, A.J.: A scalable memory model for low-level code. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 290–304. Springer, Heidelberg (2009)
19. Reynolds, J.C.: Intuitionistic reasoning about shared mutable data structure. In: Millenial Perspectives in Computer Science (2000)

Automatic Generation of Inductive Invariants from High-Level Microarchitectural Models of Communication Fabrics

Satrajit Chatterjee and Michael Kishinevsky

Intel Corporation, Hillsboro OR 97124, USA
satrajit.chatterjee@intel.com

Abstract. Abstract microarchitectural models of communication fabrics present a challenge for verification. Due to the presence of deep pipelining, a large number of queues and distributed control, the state space of such models is usually too large for enumeration by protocol verification tools such as Murphi. On the other hand, we find that state-of-the-art RTL model checkers such as ABC have poor performance on these models since there is very little opportunity for localization and most of the recent capacity advances in RTL model checking have come from better ways of discarding the irrelevant parts of the model. In this work we explore a new approach for verifying these models where we capture a model at a high level of abstraction by requiring that it be described using a small set of well-defined microarchitectural primitives. We exploit the high level structure present in this description, to automatically strengthen some classes of properties, in order to make them 1-step inductive, and then use an RTL model checker to prove them. In some cases, even if we cannot make the property inductive, we can dramatically reduce the number and complexity of lemmas that are needed to make the property inductive.

1 Introduction

Consider the microarchitectural model shown in Figure 1. It consists of a source that non-deterministically generates packets that contain the 6-bit value 0. The source feeds into a pair of serially connected FIFOs each of size k, the second of which feeds into a sink that consumes a packet non-deterministically. The communication between the source, the FIFOs and the sink is by means of a simple handshake. We present a formal semantics for these microarchitectural primitives in Section 3, but we hope that for now this intuitive description suffices.

Consider the problem of verifying that any packet seen at the output of the second FIFO contains the value 0. If we generate Verilog from this description and use a state-of-the-art RTL model checking engine such as ABC [3] (winner of the 2008 CAV Hardware Model Checking contest), we find that this apparently trivial problem is surprisingly hard even for small values of k. For instance, even for $k = 4$, ABC takes about 10 minutes to solve this problem on an Intel 3 GHz Xeon processor resorting to interpolation to prove it. Our experience

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 321–338, 2010.

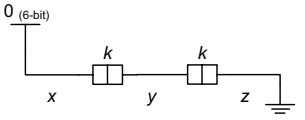


Fig. 1. A simple microarchitectural model with a source that generates the 6-bit value 0, two queues that can store k elements each and a sink. The components are connected by channels x, y and z.

with other industrial tools is similar. And this is for a system with only two queues and a simple topology. In our work on modeling the microarchitecture of communication fabrics we routinely encounter systems where a packet may traverse tens of queues in its lifetime (due to pipelining, path splitting and reconvergence, etc.) and there is complex control logic for resource management. Therefore, even if each queue is sized minimally and packets are represented abstractly, there is still a lot of state. RTL model checkers – though useful for bounded model checking – are unsuccessful in producing proofs for all but the simplest examples even when run for days or weeks. On the other hand, explicit state model checkers such as Murphi run out of memory since there are many interleavings due to non-determinism and deep pipelining.

If we go back to our example, it is obvious to a human designer that the property should hold. It is obvious since we are able to use our knowledge of queues in order to reason about the system. However, when we throw this problem to ABC or Murphi, this high-level information is lost. ABC sees a sea of gates, and Murphi a sea of rules. The traditional approach to handling such verification problems is to resort to theorem proving, or its cousin manual invariant strengthening. In manual invariant strengthening, a verification engineer adds additional invariants (called lemmas) to the model so that the entire set of invariants becomes inductive. Adding these additional invariants is a black art often requiring expertise both in formal verification and the system being verified [10].

In this work, we seek a less labor-intensive way of exploiting the high-level structure of our models than theorem proving or invariant strengthening. The key idea is to require that the microarchitectural models be described in terms of a small set of primitives such as queues, arbiters, forks and joins. Using our knowledge of these primitives, we can *automatically* add a number of lemmas so that the whole set of invariants becomes (1-step) inductive. Most of these lemmas are not local primitive-specific invariants, but are obtained by global analysis of the model. The experimental results are very encouraging: with no or little human effort and little CPU time, we can now prove a number of properties on real models which could not be proved before. In our example above, all necessary lemmas are added automatically, and ABC discharges the resulting problem in almost no time.

The requirement that the model be expressed in terms of specific primitives could be a difficult one to satisfy in general. However, the set of primitives we use in this work originated in a project aimed at reducing the effort required to write

microarchitectural models of communication fabrics [4]. Using this modeling methodology we have been able to capture the microarchitecture of a number of real designs and to validate them using simulation and bounded model checking. The goal of this work is to extend verification to obtain full proofs of correctness for some important types of properties.

The use of "high-level structure" for more efficient model checking is a holy grail of hardware verification. We believe this work makes a contribution in that direction by presenting a concrete proposal for describing hardware at a level of abstraction higher than RTL along with a couple of analysis techniques that illustrate how such structure could be exploited for efficient verification. The properties we consider are simpler than those verified in previously published manual efforts (e.g. see [9,10] and references therein) but we seek more automation. On the other hand, the use of high-level structure allows us to infer invariants which would be very difficult for existing automatic RTL-based methods (e.g. see [1] and references therein) to discover. What we present is only a beginning, and we hope that these techniques can be extended to an even larger class of properties in future work.

2 Methodology

Our microarchitectural models are described by instantiating components from a library of primitives and connecting them. We refer to these models as xMAS networks (xMAS stands for eXecutable MicroArchitectural Specification). The properties to be verified are specified on these networks. For verification, an xMAS network is compiled down into a synchronous model (single clock, edge-triggered Verilog to be precise) which is then verified. We refer to this model as the *synchronous model*.

Although the techniques presented in this paper could be used to directly verify xMAS models instead of the synchronous models, in this work, we simply use the high-level structure in the xMAS models to discover new invariants which are then used in the verification of the synchronous model. We choose this approach partly for engineering convenience (we use a conventional model checker as the trusted engine and view the analysis described in this paper as providing verification hints) and partly because the methods described in this paper cannot be used to prove all properties of interest (in particular liveness). A nice side effect of this approach is that the invariants we add get checked by the model checker rather than being assumed as given.

3 xMAS Models

xMAS models are constructed by instantiating components from a library of microarchitectural primitives and connecting them with *channels*. Channels are typed. In the synchronous model, a channel x with type α has two boolean signals x.irdy (for initiator ready) and x.trdy (for target ready) for control and one signal x.data that has type α for the data.

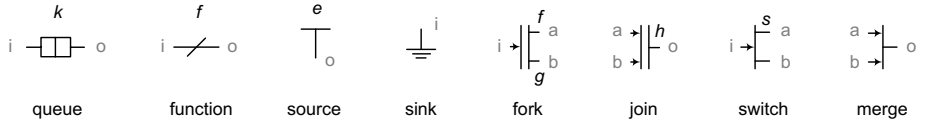


Fig. 2. A key showing the symbols for the various primitives used to model microarchitectural blocks. Section 3 describes these components in detail. The italicized letters (k, f, e, g, h and s) indicate parameters. Whenever we use these primitives in a diagram we need to specify values for these parameters. Often, to avoid clutter we do not show these values explicitly trusting that they are clear from the context. In contrast, the gray letters (i, o, a, and b) in this figure only indicate port names and are only shown to help you understand the formal definitions in Section 3. Observe that for some components such as the fork, we place the parameter close to the "corresponding" port in the diagram.

A channel is connected to exactly two components: one component called the *initiator* that "writes" to the channel (via an *output* port) and another component called the *target* that "reads" from the channel (via an *input* port). In the synchronous model, the initiator drives irdy and data signals (and reads trdy) whereas the target drives trdy (and reads irdy and data). Intuitively, a data element (or a packet) is transferred across a channel in those cycles when both irdy and trdy are true. Note that a channel is just a three wires and stores no state. A channel is represented in our diagrams by a line.

An xMAS network may be viewed as a directed graph with the components as nodes and channels as edges. Edges are directed from initiator to target.

Example. In Figure 1, there are three channels x, y and z. For channel x, the initiator is the source and the target is the first queue. Thus x is connected to the output port of the source and to the input port of the first queue. The output port of the first queue is connected to channel y.

Figure 2 shows the library of kernel primitives. We formally specify each primitive by providing the synchronous equations that are generated for it. We present this in some detail because the exact definitions are important to understand the invariants that we generate later. These definitions may be skimmed on a first reading.

Queue. In our models, storage is implemented by queues.[1] In terms of interface, a queue is one of the simplest primitives. It is parameterized by a type α of the elements stored in the queue and a non-negative integer k that indicates the capacity of the queue. It has one input port i which is connected to the target end of a channel that is used to write data into the queue. Clearly, this channel must have type α, and for convenience we say that port i also has type α, denoted by $i : \alpha$. Likewise, the output port $o : \alpha$ is connected to the

[1] Our queues are always FIFO i.e. first-in-first-out.

initiating end of the channel that reads data out of the queue. The equations for a queue are:

$$\text{o.irdy} := (\textbf{pre}(\text{num}) \neq 0) \qquad \text{i.trdy} := (\textbf{pre}(\text{num}) \neq k)$$
$$\text{enq} \quad := \text{i.irdy } \textbf{and } \text{i.trdy} \qquad \text{deq} \quad := \text{o.irdy } \textbf{and } \text{o.trdy}$$

where enq and deq are combinational signals defined for convenience, and num is the current occupancy of the queue given by:

$$\begin{aligned} \text{num} := \; & \textbf{pre}(\text{num}) + 1 \; \textbf{if } \text{enq } \textbf{and not } \text{deq} \\ & \textbf{pre}(\text{num}) - 1 \; \textbf{if } \text{deq } \textbf{and not } \text{enq} \\ & \textbf{pre}(\text{num}) \qquad \textbf{otherwise} \end{aligned}$$

where **pre** is the standard synchronous operator that returns the value of its argument in the previous cycle and the value 0 in the first cycle [2]. The elements in the queue are stored in an array called mem of size k of signals of type α. These are indexed by head and tail pointers used for reading and writing, correspondingly.

$$\begin{aligned} \text{head} &:= \textbf{if } \text{deq } \textbf{then } \textbf{inc}_k(\textbf{pre}(\text{head})) \; \textbf{else } \textbf{pre}(\text{head}) \\ \text{tail} &:= \textbf{if } \text{enq } \textbf{then } \textbf{inc}_k(\textbf{pre}(\text{tail})) \; \textbf{else } \textbf{pre}(\text{tail}) \end{aligned}$$

where $\textbf{inc}_k(x) \equiv \textbf{if } x = k-1 \textbf{ then } 0 \textbf{ else } x+1$. For $j \in \{0, k-1\}$ we have

$$\text{mem}_j := \textbf{if } \text{enq } \textbf{and } j = \textbf{pre}(\text{tail}) \textbf{ then } \text{i.data } \textbf{else } \text{pre}(\text{mem}_j)$$

and,

$$\begin{aligned} \text{o.data} := \; & \textbf{pre}(\text{mem}_0) && \textbf{if pre}(\text{head}) = 0 \\ & \textbf{pre}(\text{mem}_1) && \textbf{if pre}(\text{head}) = 1 \\ & \vdots \\ & \textbf{pre}(\text{mem}_{k-1}) && \textbf{if pre}(\text{head}) = k-1 \end{aligned}$$

Among our set of primitives a queue is the only one that can store data. It is also the only delay element: even if the queue is empty, an input packet is visible at the output only after 1 cycle.

Source. A *source* is a primitive which is parameterized by a constant expression $e : \alpha$.[2] Each cycle, it non-deterministically attempts to send a packet e through its output port. A source has a single output port $o : \alpha$ and is governed by the following equations:[3]

$$\text{o.irdy} := \text{oracle } \textbf{or pre}(\text{o.irdy } \textbf{and not } \text{o.trdy}) \qquad \text{o.data} := e$$

where oracle is an unconstrained primary input that is used to model the non-determinism of the source in the synchronous model. Each source has its own oracle. We define o.irdy in this specific manner to keep it persistent regardless of the oracle behavior: i.e. once a source makes a value available on the channel, it preserves that value until a transfer. Also note that one can imagine more complex sources which emit arbitrary values from a given set. However, for ease of exposition we stick to the simpler definition above.

[2] Henceforth we only mention the value parameters of a component and leave the type parameters implicit.

[3] When o.irdy is false, o.data is a don't care. But for brevity in the equations, we always assign to o.data rather than only when o.irdy is asserted.

Sink. Dually, a sink is a component which non-deterministically consumes a packet. It has one input port $i : \alpha$ and is characterized by the following equation:

i.trdy := oracle **or pre**(i.trdy **and not** i.irdy)

Function. A *function* primitive is used to model transformations on the data. It is parameterized by a function $f : \alpha \to \beta$. It has an input port $i : \alpha$ and an output port $o : \beta$ and is fully characterized by the following equations:

o.irdy := i.irdy o.data := f(i.data) i.trdy := o.trdy

Note that f is a combinational function that is applied to the input data to generate the output data.

Fork. A *fork* is a primitive with one input port $i : \alpha$ and two outputs ports $a : \beta$ and $b : \gamma$ parameterized by two functions $f : \alpha \to \beta$ and $g : \alpha \to \gamma$. Intuitively, a fork takes an input packet and creates a packet at each output. It coordinates the input and outputs so that a transfer only takes place when the input is ready to send and both the outputs are ready to receive. Formally,

a.irdy := i.irdy **and** b.trdy a.data := f(i.data)
b.irdy := i.irdy **and** a.trdy b.data := g(i.data)
i.trdy := a.trdy **and** b.trdy

Join. A *join* is the dual of a fork. It has two input ports $a : \alpha$ and $b : \beta$ and one output port $o : \gamma$. It is parameterized by a single function $h : \alpha \times \beta \to \gamma$. Intuitively, a join takes two input packets (one at each input) and produces a single output packet. It coordinates the inputs and output so that a transfer only takes place when the inputs are ready to send and the output is ready to receive. Formally,

a.trdy := o.trdy **and** b.irdy b.trdy := o.trdy **and** a.irdy
o.irdy := a.irdy **and** b.irdy o.data := h(a.data, b.data)

Switch. A *switch* is a primitive to route packets in the network. It has an input port i and two output ports a and b, all of type α. It is parameterized by a switching function $s : \alpha \to$ Bool. Informally, the switch applies s to a packet x at its input, and if $s(x)$ is true, it routes the packet to port a, and otherwise it routes it to port b. Formally,

a.irdy := i.irdy **and** s(i.data) a.data := i.data
b.irdy := i.irdy **and not** s(i.data) b.data := i.data
i.trdy := (a.irdy **and** a.trdy) **or** (b.irdy **and** b.trdy)

Merge. Arbitration is modeled by a *merge* primitive that selects one packet among multiple competing packets. A merge has multiple input ports and one output port. Requests for a shared resource are modeled by sending packets to a merge, and a grant is modeled by the selected packet. For simplicity we present here a complete definition of a two-input merge that has two input ports $a : \alpha$ and $b : \alpha$ and one output $o : \alpha$.

o.irdy := a.irdy **or** b.irdy
o.data := a.data **if** u **and** a.irdy
b.data **if not** u **and** b.irdy
a.trdy := u **and** o.trdy **and** a.irdy
b.trdy := **not** u **and** o.trdy **and** b.irdy

where u is a local Boolean state variable to ensure fairness. We could choose a specific fairness algorithm such as

$u :=$ 1 **if** a.irdy **and not** b.irdy
0 **if not** a.irdy **and** b.irdy
not pre(u) **if pre**(o.irdy **and** o.trdy)
pre(u) **otherwise**

Example. Figure 3 shows two agents P and Q communicating via a router. Packets are modeled by triples (t, s, d), where $t \in \{\text{req}, \text{rsp}\}$ is the type of the packet, $s \in \{P, Q\}$ is the source and $d \in \{P, Q\}$ is the destination. Each agent creates new requests for the other agent or for itself. When an agent receives a request (from the other agent or from itself) it produces a response by changing the type of the message and swapping the source and the destination. The response is produced after a non-deterministic delay. The response is sent back to the requester where it is sunk after a non-deterministic delay. The router routes messages according to their destinations i.e. d. (In practice this simplified microarchitecture would not be used since it deadlocks. Deadlocks can be avoided by using virtual channels as we discuss later.)

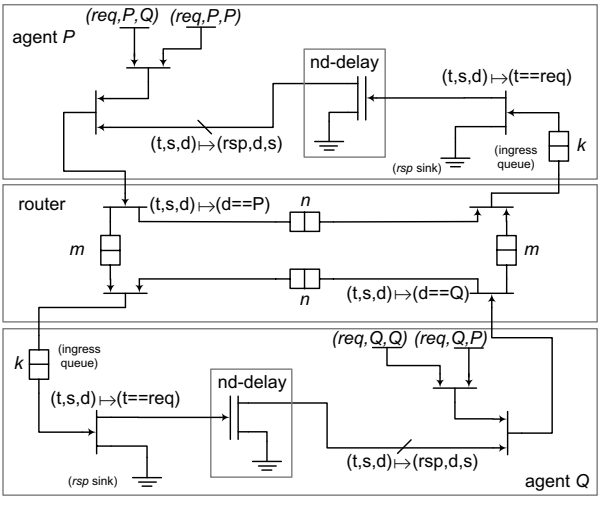


Fig. 3. Example showing a pair of agents communicating over a simple fabric (see text for details). The nd-delay box models non-deterministic delay (the functions of the fork are identity). Since each symbol has a precise formal semantics (see Section 3) this figure is a precise executable description.

4 Analysis for Channel Properties

A very common verification problem on xMAS networks is to check that all values flowing through a channel satisfy some property. For instance, at the input of an agent, we may wish to check that all packets that arrive have the agent as the destination. Invariants of this kind are called channel properties, and in this section we see how such invariants may be strengthened.

4.1 Channel Properties

If x is a channel that has type α, a *channel property* is a function $p : \alpha \to \{0, 1\}$. Intuitively, if a property p is asserted on a channel x, it means that whenever a valid value is seen on the channel (i.e. x.irdy is asserted), the data on the channel must satisfy p. Formally, a channel property p on a channel x corresponds to the LTL invariant $\mathbf{G}(x.\text{irdy} \implies p(x.\text{data}))$ in the synchronous model. For brevity, we sometimes simply say property instead of channel property.

Example. The verification problem in the introduction corresponds to verifying the channel property $v \mapsto (v = 0)$ on channel z.[4] This corresponds to the LTL property $\mathbf{G}(z.\text{irdy} \implies (z.\text{data} = 0))$ in the synchronous model.

4.2 Propagating Channel Properties

Given a channel property p, we can derive properties on other channels that are "implied" by p using a set of rules. These rules are similar in spirit to Hoare rules [8] used in program verification and are derived syntactically (i.e. no reasoning is involved). The goal is to strengthen the LTL invariant corresponding to p in the synchronous model with the additional invariants obtained from the new channel properties. The soundness of these rules may be verified from the definitions given in Section 3.

Rule for Queue. Since a queue does not modify the data it holds, a property holds on the output of a queue iff it holds on the input.

Example. In our running example (Figure 1), the property $v \mapsto (v = 0)$ holds at z iff it holds at y. Similarly the property holds at y iff it holds at x. It turns out that adding the LTL properties corresponding to the channel properties for x and y, does not make the resulting verification problem on the synchronous model inductive. We need further strengthening, and we return to this topic shortly.

Rule for Function. Given an instance of a function primitive with the parameter $f : \alpha \to \beta$, a channel property p holds at the output iff the property $p' = p \circ f$ holds at the input.

Rule for Switch. Consider an instance of a switch whose switching function is $s : \alpha \to \beta$. The channel property p holds at the output a iff the property

[4] By "$v \mapsto (v = 0)$" we mean the function that is 1 iff the input is equal to 0, i.e. the function $\lambda v.(v = 0)$ using λ notation.

$v \mapsto (s(v) \implies p(v))$ holds at the input. Likewise, a property p holds at output b iff the property $v \mapsto ((\neg s(v)) \implies p(v))$ holds at the input.

Rule for Mux. A channel property holds on the output iff it holds on each input.

Rule for Fork. A channel property p holds on the output a of a fork iff $p' = p \circ f$ holds on the input. Similarly, p holds on the output b of a fork iff $p' = p \circ g$ holds on the input.

Rule for Restricted Join. Propagating a property across a join is tricky since the output of a join in general could be functionally dependent on both inputs. However, in our examples drawn from the domain of communication fabrics, joins are only used to control access to resources (e.g. see examples of credit logic and virtual channels in Section 5). Therefore, the join function depends only on at most one input of the join (called the *functional* input) i.e. it is of the form $h : \alpha \to \gamma$ (instead of $h : \alpha \times \beta \to \gamma$). In such cases the other input carries tokens (i.e. values having the unit type). It is easy to detect such joins automatically since the join function h syntactically depends only on one of the inputs. If h is constant, then either input may be taken as the functional input. Given such a join with the restricted function $h : \alpha \to \gamma$, a property p holds at the output iff $p' = p \circ h$ holds at the functional input of the join. Extending propagation to general joins appears to be a hard problem since it involves reasoning about multiple channels.[5]

4.3 Queue Invariants

If we have a channel property p at the output of a queue, using the rule for queues presented above, we also have the property p at the input of the queue. However, simply adding the invariants from these properties to the LTL model does not make the synchronous problem (1-step) inductive. It is easy to see why: Suppose a queue is in a state where it has more than 2 elements. Even if these properties hold at the output and input of the queue, at best they guarantee that only the oldest and youngest element in the queue satify p. They say nothing about the other elements in the queue.

Therefore we need additional invariants to ensure that *every* element stored in the queue satisfies p. For $j \in [0, k)$, where k is the size of the queue, we add the LTL invariant (recall the state variables of a queue from Section 3)

$$\mathbf{G}(\text{used}_j \implies p(\text{mem}_j))$$

where used_j is a predicate over the state that indicates if the jth storage element in the queue is used or not. It is defined as follows:

$$\begin{aligned}\text{used}_j := &(\text{head} < \text{tail} \ \textbf{and} \ (\text{head} \leq j \ \textbf{and} \ j < \text{tail})) \ \textbf{or} \\ &(\text{head} > \text{tail} \ \textbf{and} \ (\text{head} \leq j \ \textbf{or} \quad j < \text{tail})) \ \textbf{or} \ (\text{num} = k)\end{aligned}$$

[5] Even with general joins there is an easy case. If p is a property such that $p' = p \circ h$ depends on only one variable, then it suffices to propagate p' along the corresponding input.

Along with this, we add the LTL assertions $\mathbf{G}(\text{num} \leq k)$, $\mathbf{G}(\text{head} < k)$ and $\mathbf{G}(\text{tail} < k)$ to ensure that these state variables are within bounds. Finally, we need to add the following invariants to establish the correct relationship between these 3 state variables:

$$\mathbf{G}(\text{head} < \text{tail} \implies \text{head} + \text{num} = \text{tail})$$
$$\mathbf{G}(\text{head} > \text{tail} \implies \text{head} + \text{num} = \text{tail} + k)$$
$$\mathbf{G}(\text{head} = \text{tail} \implies \text{num} = 0 \textbf{ or } \text{num} = k)$$

These assertions are used to ensure that the head and tail pointers behave as expected and provide a local over-approximation of the state-space.

4.4 A Note on Local Invariants

The queue invariants added above block off portions of the unreachable state space that would otherwise lead to false counter examples in induction. However since these invariants are local, any correlation between different queues is not captured. However, this is exactly how a human designer thinks about the system: for example seldom would the correctness of a design depend on say two head pointers in two different queues taking on the same value in all portions of the reachable state space.

Indeed, if the correct operation of a design relies on the correlation between different components, typically this is enforced in the design by some explicit communication structure between them. A common case in our models for this case is when the occupancy of multiple queues are correlated in the reachable state space. We study this problem in the next section where we follow this communication trail to infer the appropriate invariants.

The queue is our main state holding element. Among all the primitives, the only other interesting state holding element is the merge which maintains state for fairness. If the merge has multiple inputs, then the appropriate local invariants for the fairness logic need to be added. (For the particular 2-input merge presented in Section 3, we do not need to add constraints for the u variable since it can take on both 0 and 1 values in the reachable space.)

4.5 Propagation Algorithm

Given a property p on a channel, we try to maximally propagate it backwards using the obvious iterative algorithm. This is done by looking at the initiator of the channel, and applying the corresponding rule from Section 4.2. This creates new properties at the inputs of the initiator. This process is repeated for each newly added property. If the initiator is a source, then the property is not (cannot be!) propagated further. If the xMAS network has a directed cycle, the above process will not terminate. We handle this by recording the "parent" and stopping when a cycle is encountered.

After all properties have been propagated in this manner, for each queue in the system, we add the local invariant according to the scheme described in Section 4.3 for each property at the output of the queue.

Theorem 1 (Partial Completeness). *Given an acyclic* xMAS *network* $\mathcal{N}$ *where all joins are restricted, and a property* p *on a channel in* $\mathcal{N}$ *that holds, the above algorithm adds sufficiently many invariants to make the synchronous problem 1-step inductive.*

The propagation algorithm often leads to the creation of a large number of properties. However, many properties can be discharged locally i.e. during the propagation process, they become tautologies i.e. the constant 1 function. Therefore we use a reasoning engine to detect tautological properties and do not propagate them further. This is an important optimization in practice.

Example. In the example of Figure 3, let l be the property $(t, s, d) \mapsto (d = P)$ at the ingress queue of agent P. If we propagate l backwards through the queues and switches in the router using the above algorithm, we find that the properties that are obtained from l at each input of the router are of the form $(t, s, d) \mapsto ((d = P) \implies (d = P))$ which is a tautology. These tautological properties need not be propagated further.

Remark on cycles. Most properties become tautologies during propagation, so cycles in the xMAS network are not a problem (as in the example above). However, for those that do not become tautologies, it may be necessary to add additional channel invariants on loops to "break" the cycle. Furthermore, in many cases in communication fabrics we find that packets loop at most k times, where k is small (i.e. 1 or 2). For example in Figure 3, $k = 2$ (for a self-request). We could handle such cases automatically by unrolling loops $k + 1$ times.

5 Invariants from Flows

As remarked in Section 4.4, if the correct operation of a design relies on correlation between state variables in different components then in a real design there is usually an explicit communication mechanism between them for coordination. In this section we present an algorithm to analyze a commonly-occuring communication of this form that leads to correlation among the occupancies of different queues in the system. The invariants added by this analysis allow us to prove an important class of safety properties that check that the queues in a system are sized correctly. Such safety properties are necessary for reasoning about liveness.

5.1 The Basic Idea

Example (credits). Consider the xMAS network shown in Figure 4 which shows a master agent M communicating with a target T. The credit logic portion of T issues at most k outstanding credits to M at any given time. Credits are modeled as values of the unit type called *tokens*. M has to wait for a credit before it can send a request to T. The purpose of this mechanism is to ensure that there is always room in T's ingress queue for requests from M i.e. nothing

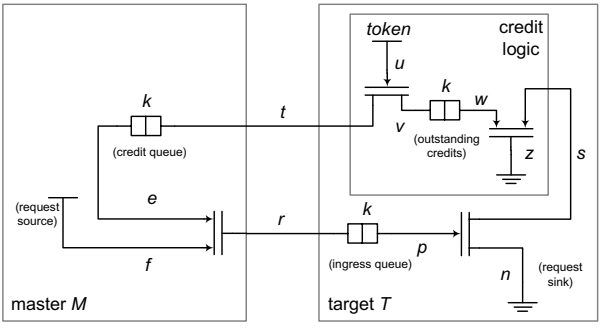


Fig. 4. Credits introduce correlation between the occupancies of different queues. Both joins are restricted in the sense of Section 4.2 since at least one input is a token.

gets stuck on channel r. Thus r is *non-blocking* i.e. satisfies the LTL property: $\mathbf{G}\ (r.\text{irdy} \implies r.\text{trdy})$. Credits are freed up when data is read from the ingress queue of T.

The non-blocking property on r is not inductive. However, by adding the invariant

$$\mathbf{G}(\text{num}_c + \text{num}_i = \text{num}_o)$$

to the synchronous model, the problem becomes inductive.[6] Here, num_c is the num variable of the credit queue in M, num_i the same for the ingress queue in T and num_o for the outstanding credits queue in T.

The question now is how can we detect such global assertions automatically? If x is a channel, let λ_x denote the number of packets that have been transferred on x upto a given point in time (i.e. λ_x is the count of the number of cycles so far in which $x.\text{irdy}$ and $x.\text{trdy}$ were both asserted). Now, from the equations of a join in Section 3 it is easy to see that either a transfer happens on both inputs and the output of a join or there is no transfer at any input or the output. Thus for the two joins in Figure 4 we have,

$$\lambda_e = \lambda_f = \lambda_r \qquad \text{and} \qquad \lambda_s = \lambda_w = \lambda_z.$$

Similarly, for a fork it can be verified that either a transfer happens on both output and the input or there is no transfer at all. Thus for the two forks in the system we have the equations

$$\lambda_u = \lambda_t = \lambda_v \qquad \text{and} \qquad \lambda_p = \lambda_n = \lambda_s.$$

A queue is more interesting. Any packet that enters a queue is either still in the queue or has exited through the output channel. Thus from the three queues in Figure 4 we get the following three equations:

$$\lambda_r = \text{num}_i + \lambda_p \qquad \lambda_t = \text{num}_c + \lambda_e \qquad \lambda_v = \text{num}_o + \lambda_w$$

[6] Assuming that the (local) assertions $\mathbf{G}(\text{num} \leq k)$ for each queue have already been added.

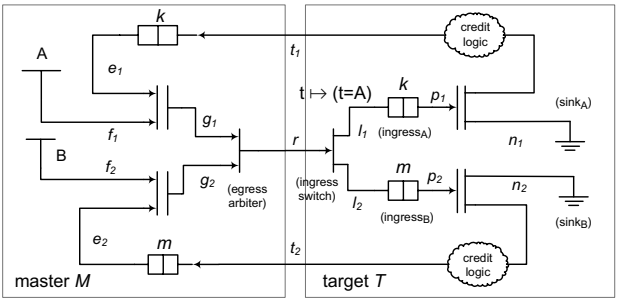


Fig. 5. Example of a shared communication path that requires more precise flow analysis. The credit logic bubbles encapsulate the logic shown in the credit logic box of Figure 4. The switch in the target routes A packets to l_1 and B packets to l_2. The joins in M are restricted and have the identity function.

From these 7 equations, we can eliminate the λ variables to get the desired relationship between the num variables. This can be done automatically in the following manner. First we create a matrix from the equations where all λ variables are to the left and all num variables are to the right. Then this matrix is converted to Reduced Row Echelon (RRE) form by Gaussian elimination (over the rationals). Finally, we select the equations from the RRE form which involve only the num variables (i.e. the coefficients of all λ variables are 0).

Note that the λ variables are unbounded and by this elimination process, we are only left with relations in only the num variables (which are bounded by the size of the queues). Hence these relations can be added as invariants to the synchronous model.

The technique described in this section resembles generation of place invariants in Petri nets [5]. However, rather than modeling the communication fabric with Petri nets (which leads to an overhead of using explicit back-pressure arcs and complexity in modeling the data-path) we derive those invariants directly from more compact and natural xMAS specifications.

5.2 Shared Communication

In the presence of shared communication channels the approach presented above needs to be refined.

Example (virtual channels). Virtual channels lead to sharing. They are commonly used in communication fabrics to multiplex multiple logical streams onto a single physical link with the guarantee that even if one stream is blocked at the receiver, the other streams still make progress [7].

Figure 5 shows a simple example of virtual channels. A master agent M sends two types of messages A and B (think of these as perhaps requests and responses) to a target T over a single channel r. The ingress switch in T routes A and B packets to their respective ingress queues. The credit pattern of Figure 4 is used

to ensure that whenever a packet is presented to the egress arbiter of M, there is guaranteed to be room in the corresponding ingress queue in T. Thus channel r is non-blocking.

Once again, the non-blocking property on r is not inductive. However, if we add the invariants $\mathbf{G}(\text{num}_{c_A} + \text{num}_{i_A} = \text{num}_{o_A})$ and $\mathbf{G}(\text{num}_{c_B} + \text{num}_{i_B} = \text{num}_{o_B})$ for each credit loop, then the problem becomes inductive. Here, num_{c_A} refers to the num variable of the credit queue in M associated with the A packets, and so on. However, if we try the approach from the previous example (with suitable extensions for muxes and switches) we find that we can only derive the weak invariant: $\mathbf{G}(\text{num}_{c_A} + \text{num}_{i_A} + \text{num}_{c_B} + \text{num}_{i_B} = \text{num}_{o_A} + \text{num}_{o_B})$ which is not enough to prove the property.

We can improve the precision of the analysis by defining a λ variable *per flow* through a channel. For example we know that two types of values flow through channel r. Therefore we introduce two variables λ_r^A and λ_r^B for r where λ_r^A is a count of the number of cycles when r.irdy and r.trdy have been asserted *and* r.data was equal to A. Similarly, λ_r^B for B. Since there are two flows through r, we assume that two flows are possible through g_1 and g_2 and through f_1 and f_2 and associate two λ variables from each of these channels: one for A and one for B. For all the other channels, we associate only a single λ variable since there are only single flows through them. (We will see later how to automatically figure out the number of flow variables needed.)

Since the ingress switch in T routes A to channel l_1 and B to channel l_2, we have

$$\lambda_r^A = \lambda_{l_1} \qquad \text{and} \qquad \lambda_r^B = \lambda_{l_2}$$

For a merge, a packet at the output must come from one or the other input. Therefore, we have the following equations for the egress arbiter in M:

$$\lambda_{g_1}^A + \lambda_{g_2}^A = \lambda_r^A \qquad \text{and} \qquad \lambda_{g_1}^B + \lambda_{g_2}^B = \lambda_r^B$$

Observe that one input of each join in M is a token input i.e. the joins are restricted. We have the following relations between the outputs and the functional inputs:

$$\lambda_{f_1}^A = \lambda_{g_1}^A \qquad \lambda_{f_1}^B = \lambda_{g_1}^B \qquad \lambda_{f_2}^A = \lambda_{g_2}^A \qquad \lambda_{f_2}^B = \lambda_{g_2}^B$$

and the following relations between the token inputs and the outputs:

$$\lambda_{e_1} = \lambda_{g_1}^A + \lambda_{g_1}^B \qquad \lambda_{e_2} = \lambda_{g_2}^A + \lambda_{g_2}^B$$

Each source however generates only one type of packet. Therefore we can set the other λ variable on the output channel to zero i.e. $\lambda_{f_1}^B = 0$ and $\lambda_{f_2}^A = 0$.

All the other components only interface with channels carrying single flows, and we add equations as in the credit example. Finally, as before, by eliminating the λ variables using Gaussian elimination, we obtain the desired relations among the num variables.

5.3 Algorithm for Discovering Flow Invariants

Formally, if x is a channel that has type α, a *flow* on x is a function $p : \alpha \to \{0, 1\}$. (Note the similarity with channel properties.) Our goal is to compute the set of flows for each channel and the equations relating the λ variables for these flows.

Step 1. Sort the xMAS graph in reverse "topological" order starting from the sinks using the textbook depth-first-search (DFS) based topological sort algorithm [6, §22.4]. If the xMAS network is cyclic, this has the effect of topologically sorting the DAG obtained by deleting the backedges in the DFS.

Step 2. Assign the constant 1 function as the flow on the inputs to the sinks and on the backedge channels. Now we process each component in the network in the reverse "topological" order computed above by applying the following rules to propagate flows (we use the same parameter names as in Figure 2 and use port names to refer to the corresponding channels):

Queue. For each flow p on the output channel o, we create a new flow p on the input channel i. We also add a new state variable called num^p to the queue that tracks how many elements satisfying p are currently in the queue. We also add an assertion that equates num^p to the number of elements that satisfy $(\text{used}_j \implies p(\text{mem}_j))$ in the queue. Finally, we add the equation $\lambda_i^p = \text{num}^p + \lambda_o^p$.

Function. For each flow p on the output channel o, we create a new flow $p' = p \circ f$ on the input channel i and add the equation $\lambda_i^{p'} = \lambda_o^p$.

Switch. For each flow p on the output channel a, we create a flows $p' = v \mapsto (s(v)\ \textbf{and}\ p(v))$ on the input channel i and add the equation $\lambda_i^{p'} = \lambda_a^p$. Similarly for flows on output b.

Merge. For each flow p on the output channel o, we create a flow p on input a and another flow p on input b and add the equation $\lambda_a^p + \lambda_b^p = \lambda_o^p$.

Fork. For each pair (p, q) where p is a flow on output a and q on output b, we create a new flow $r = v \mapsto (p(f(v))\ \textbf{and}\ q(g(v)))$ on input. For each flow p on output a, we add the equation $\lambda_a^p = \Sigma_r \lambda_i^r$ where r ranges over flows that were added to i due to p. Similar equations are added for each flow on b.

Join. Once again, we limit our attention to restricted joins. For each flow p on the output channel o, we add a flow $p' = p \circ h$ to the functional input (suppose it is the input a). We add the constant 1 flow to the other input (i.e. b) and the equations $\lambda_a^{p'} = \lambda_o^p$ and $\lambda_b^1 = \Sigma_p \lambda_o^p$ where p ranges over all the flows on o.

Source. For each flow p in the output o, we check if $p(e)$ is true or not. If $p(e)$ is false, then we add the equation $\lambda_o^p = 0$ and mark p as *dead*.

During the above process, each time a new flow is created, we record its parent(s). Furthermore, if a new flow is unsatisfiable i.e. the constant 0 function we mark it dead and do not propagate it further.

Step 3. If the xMAS DAG is cyclic, for each channel x that is a backedge, the above propagation process adds new flows. These need to be related to the constant 1 flow which was assigned before starting propagation. Therefore we add $\lambda_x^1 = \Sigma_p \lambda_x^p$ where p ranges over all the flows added to x during propagation.

Step 4. If all children of a flow p at a channel x are dead, we mark x as dead as well and add the equation $\lambda_x^p = 0$. We repeat this process until no new flows can be marked dead.

Theorem 2 (Inductivity). *The set of equations obtained by this process is an inductive invariant of the synchronous model.*

Step 5. Finally, the λ variables are eliminated as explained before to obtain relations between the num variables. Note that it is possible that there are no relations among the num variables (e.g. Figure 1).

Remark. Since the λ variables correspond to channels which hold no state, we conjecture that eliminating them does not destroy inductivity. This has been confirmed by our experiments.

6 Experimental Results

6.1 Micro-benchmarks

Since the state-of-the-art model checking algorithms are unable to converge on any of our real examples, we present a comparison on the small examples from this paper. Table 1 shows the results of running ABC (version 91206p) on several examples (parameterized on k) without the addition of invariants as described in this paper.

The first example is from Figure 1 where each queue is of size k. In the second example we have a series of k queues (similar to Figure 1). In the third we check the property in the example of Section 4.5, but to make the example more realistic we set the the source and destination to be 2 bits wide in the packet. Fourth and fifth are self-explanatory. In the last example we add k queues on the channel r in Figure 5.

Column i in the table is the number of primary inputs (oracles); r and n are number of registers and AIG nodes (after synthesis). A depth of (m, n) means

Table 1. Comparison with interpolation on micro-benchmarks. See Section 6.1 for details. Most rows have two data points corresponding to different values of the parameter k of the corresponding example.

Description	k	i	r	n	depth	time	k	i	r	n	depth	time
Two queues of size k	3	2	49	348	(5, 12)	23	4	2	63	381	BMC 24	–
k queues of size 2	3	2	49	302	(9, 12)	76	4	2	64	396	BMC 20	–
Figure 3 with all queues sized to 2							-	8	99	659	BMC 11	–
Figure 4	8	4	14	104	(13, 9)	38	12	4	14	121	BMC 27	–
Figure 5 with all queues sized to 2							-	4	17	95	(7, 4)	40
above with k queues on r	1	4	24	135	(6, 7)	112	2	4	29	151	BMC 13	–

interpolation converged in n iterations when starting from a BMC of depth $1+m$. The time is in seconds (on a 3GHz Intel Xeon CPU) with a timeout of 300 secs indicated by a dash (and we show the final BMC depth in the previous column). Note that interpolation times out on many examples.

The first three rows correspond to examples for channel propagation. In all cases when we add the invariants as described in Section 4, ABC is able to solve the problem in no time. Even if we set $k = 100$, the first example is solved in 7 seconds, the second in 1 second and the third in 40 seconds.

The remaining rows correspond to examples for flow invariants. Again without the flow invariants, interpolation has a hard time. However, in these examples we found that BDD-based reachability could solve these quickly. In all cases in our experience, the algorithm for discovering flow invariants finds exactly those invariants that are interesting. For example, in the fifth example, there are initially 43 variables and 32 equations. After elimination, we are left with two invariants (with 6 terms) corresponding to the two credit loops as expected. The time needed for both property propagation and for detecting flow invariants is neglible.

6.2 Experience on Real Examples

We have applied the techniques described in this paper to verify a number of abstract models used to validate the microarchitecture of future designs. These are drawn from the domain of communication fabrics and are characterized by deeply pipelined logic for multi-phase transactions, presence of ordering logic and several virtual channels, and peer-to-peer traffic. Even in minimal configurations, there are tens of simultaneous transactions in flight.

As a data point, previously on one of our simpler examples we were able to obtain a proof of a critical non-blocking property[7] only by severely limiting the state space by reducing the number of simultaneous outstanding transactions an agent can issue. The proof was obtained with an explicit state model checker with maximal reachability depth of 159 in 12 hours using 17GB of memory. In contrast, using the flow analysis from Section 5 on the *original* model, 16 flow relations are discovered (from an initial set of 176 equations on 220 variables) and ABC solves the resulting problem in 4.5 sec.

A big advantage of this technique is its robustness and scalability. Rather than be limited to minimal configurations (and consequently reduced concurrency), we can now verify more realistic models. Channel property verification is robust and most properties are discharged automatically. For a few properties we need to add additional channel properties to break loops. However, these invariants are natural and easy to add since they only talk about data and do not involve control at all. Finally, although it may appear that flow invariants could lead to scalability problems, so far we have not encountered any problems, even on our larger examples with dozens of flow invariants, many with tens of terms. For such problems, an inductive engine that assumes all invariants in one cycle and then checks each invariant separately in the following cycle appears to be scalable.

[7] To check for adequate buffering to avoid deadlocks.

7 Conclusion and Future Work

The concrete proposals for capturing and exploiting high-level information in hardware models presented in this work have proved very useful in practice allowing us to prove with little computational effort many hard sequential properties on real microarchitectural models which could not be proved before. The benefit seems to be in separating control from data and exploiting knowledge of the control to reduce the problem to a combinational one on the data.

The invariants we add may be seen as providing a *bag* abstraction for queues. Although the bag abstraction has proven adequate to handle our current examples (systems with restricted joins), it appears insufficient to handle systems with general joins. It would be interesting to extend the methods of this paper to reason about such systems.

Finally, a lot of the computational overhead of verifying the synchronous model may be eliminated by switching to an axiomatic semantics for xMAS models for a more direct verification. This may also be an interesting direction for building bridges to the RTL implementation.

Acknowledgments

We thank Amit Goel, Alexander Gotmanov, Chandramouli Kashyap, Umit Ogras and Sayak Ray for many helpful discussions on this work and Jesse Bingham for reviewing an early draft.

References

1. Baumgartner, J., et al.: Scalable conditional equivalence checking: An automated invariant-generation based approach. In: FMCAD 2009, pp. 120–127 (2009)
2. Benveniste, A., et al.: The synchronous language twelve years later. Proc. of the IEEE 91(1), 64–83 (2003)
3. Berkeley Logic Synthesis Group, http://www.eecs.berkeley.edu/~alanmi/abc/
4. Chatterjee, S., Kishinevsky, M., Ogras, U.Y.: Quick formal modeling of communication fabrics to enable verification. In: HLDVT 2010 (to appear, 2010)
5. Colom, J.M., Silva, M.: Convex geometry and semiflows in P/T nets. In: Proc. of Appl. and Theory of Petri Nets, 79–112 (1991)
6. Corman, T.H., et al.: Introduction to Algorithms, 2nd edn. MIT Press, Cambridge (1990)
7. Dally, W.J., Towles, B.: Principles and Practices of Interconnection Networks. Morgan Kaufmann, San Francisco (2004)
8. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. of the ACM 12(10), 576580–576583 (1969)
9. Jhala, R., McMillan, K.L.: Microarchitecture Verification by Compositional Model Checking. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 396–410. Springer, Heidelberg (2001)
10. Kaivola, R., et al.: Replacing Testing with Formal Verification in Intel Core i7 Processor Execution Engine Validation. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 414–429. Springer, Heidelberg (2009)

Efficient Reachability Analysis of Büchi Pushdown Systems for Hardware/Software Co-verification

Juncao Li[1], Fei Xie[1], Thomas Ball[2], and Vladimir Levin[2]

[1] Department of Computer Science, Portland State University
Portland, OR 97207, USA
{juncao,xie}@cs.pdx.edu
[2] Microsoft Corporation
Redmond, WA 98052, USA
{tball,vladlev}@microsoft.com

Abstract. We present an efficient approach to reachability analysis of Büchi Pushdown System (BPDS) models for Hardware/Software (HW/SW) co-verification. This approach utilizes the asynchronous nature of the HW/SW interactions to reduce unnecessary HW/SW state transition orders being explored in co-verification. The reduction is applied when the verification model is constructed. We have realized this approach in our co-verification tool, CoVer, and applied it to the co-verification of two fully functional Windows device drivers with their device models respectively. Both of the drivers are open source and their original C code has been used. CoVer has proven seven safety properties and detected seven previously undiscovered software bugs. Evaluation shows that the reduction can significantly scale co-verification.

1 Introduction

Hardware/Software (HW/SW) co-verification, verifying hardware and software together, is essential to establishing the correctness of complex computer systems. In previous work, we proposed a Büchi Pushdown System (BPDS) as a formal representation for co-verification [1], a Büchi Automaton (BA) represents a hardware device model and a Labeled Pushdown System (LPDS) represents a model of the system software. The interactions between hardware and software take place through the synchronization of the BA and LPDS. The BPDS is amenable to standard symbolic model checking algorithms [2].

In this paper, we exploit the fact that hardware and software are mostly asynchronous in a system to reduce the cost of model checking. Intuitively, when hardware and software transition asynchronously (i.e. there are no HW/SW interactions), it is unnecessary to explore all the possible state transition orders. Furthermore, we prove that special cases of the transition orders preserve the reachability properties in question. Partial order reduction identifies such special transition orders, so there are fewer interleaving possibilities to be explored during model checking. We base our approach on the concept of static partial order reduction [3], where unnecessary transition orders are pruned during the construction of the verification model. During the model construction, unnecessary transition orders are largely reduced when hardware and software are asynchronous. On the other hand, all the synchronous transitions are preserved.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 339–353, 2010.

We implemented our approach in the co-verification tool CoVer and applied it to the co-verification of two fully functional Windows device drivers (C programs for which source code is publically available) with their device models. We specify the device models based on the HW/SW interface documents that are openly available. Conceptually, a driver and its device model together form a BPDS model. CoVer converts the driver and the device model into a C program and utilizes the SLAM engine [4] to check reachability properties of the program. The abstraction/refinement process is carried out by SLAM. CoVer proved seven properties and detected seven real defects in the two drivers. All of these defects can cause serious system failures including data loss, interrupt storm, device hang, etc.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 introduces the background of this paper. Section 4 presents our reachability analysis algorithm for BPDS models. Section 5 discusses how we specify a device model as well as the implementation details of CoVer. Section 6 presents the evaluation results. Section 7 concludes and discusses future work.

2 Related Work

Kurshan, et al. presented a co-verification framework that models hardware and software designs using finite state machines [5]. Xie, et al. extended this framework to hardware and software implementations and improved its scalability via component-based co-verification [6]. However, finite state machines are limited in modeling software implementations, since they are not suitable to represent software features such as a stack.

Another approach to integrating hardware and software within the same model is exemplified by Monniaux in [7]. He modeled a USB host controller device using a C program and instrumented the device driver, another C program, in such a way as to verify that the USB host controller driver correctly interacts with the device. The hardware and software were both modeled by C programs and thus are formally PDSs. However, straightforward composition of the two PDSs to model the HW/SW concurrency is problematic, because it is known, in general, that verification of reachability properties on concurrent PDS with unbounded stacks is undecidable [8].

Bouajjani et al. [9] presented a procedure to compute predecessor reachability of PDS and apply this procedure to linear/branching-time property verification. This approach was improved by Schwoon [2], which results in a tool, Moped, for checking Linear Temporal Logic (LTL) properties of PDS. A LTL formula is first negated and then represented as a BA. The BA is combined with the PDS to monitor its state transitions; therefore the model checking problem is to compute if the BA has an accepting run. The goal of the previous research was to verify software only; however, our goal is to co-verify HW/SW systems.

Our previous work [1] did not exploit the fact that hardware and software are mostly asynchronous in a system. Techniques such as partial order reduction [10] can be applied to reduce the verification complexities via the composition (Cartesian product) of the BA and LPDS. Furthermore, our co-verification implementation in our earlier work was not automatic since it depends on two abstraction/refinement engines (for hardware and software specifications respectively) that were not completely integrated.

3 Background

3.1 Büchi Automaton (BA)

A *BA* $\mathcal{B}$, as defined in [11], is a non-deterministic finite state automaton accepting infinite input strings. Formally, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, where Σ is the input alphabet, Q is the finite set of states, $\delta \subseteq (Q \times \Sigma \times Q)$ is the set of state transitions, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of final states. $\mathcal{B}$ accepts an infinite input string iff it has a run over the string that visits at least one of the final states infinitely often. A run of $\mathcal{B}$ on an infinite string s is a sequence of states visited by $\mathcal{B}$ when taking s as the input. We use $q \xrightarrow{\sigma} q'$ to denote a transition from state q to q' with the input symbol σ.

3.2 Labeled Pushdown System (LPDS)

A *LPDS* $\mathcal{P}$, as defined in [1], is a tuple $(I, G, \Gamma, \Delta, \langle g_0, \omega_0 \rangle)$, where I is the input alphabet, G is a finite set of global states, Γ is a finite stack alphabet, $\Delta \subseteq (G \times \Gamma) \times I \times (G \times \Gamma^*)$ is a finite set of transition rules, and $\langle g_0, \omega_0 \rangle$ is the initial configuration. LPDS is an extension of PDS [2] in such a way that a LPDS can take inputs. A LPDS transition rule is written as $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', w \rangle$, where $\tau \in I$ and $((g, \gamma), \tau, (g', w)) \in \Delta$. A configuration of $\mathcal{P}$ is a pair $\langle g, \omega \rangle$, where $g \in G$ is a global state and $w \in \Gamma^*$ is a stack content. The set of all configurations is denoted by $Conf(\mathcal{P})$. The head of a configuration $c = \langle g, \gamma v \rangle$ is $\langle g, \gamma \rangle$ and denoted as $head(c)$, where $\gamma \in \Gamma, v \in \Gamma^*$; the head of a rule $r : \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle$ is $\langle g, \gamma \rangle$ and denoted as $head(r)$. The head of a configuration decides the transition rules that are applicable to this configuration, where the deciding factors are the global state and the top stack symbol. Given a rule $r : \langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', \omega \rangle$, for every $v \in \Gamma^*$, the configuration $\langle g, \gamma v \rangle$ is an immediate predecessor of $\langle g', \omega v \rangle$ and $\langle g', \omega v \rangle$ is an immediate successor of $\langle g, \gamma v \rangle$. We denote the immediate successor relation in PDS as $\langle g, \gamma v \rangle \overset{\tau}{\Rightarrow} \langle g', \omega v \rangle$, where we say this state transition follows the PDS rule r. The reachability relation, $\Rightarrow^*$, is the reflexive and transitive closure of the immediate successor relation. A path of $\mathcal{P}$ on an infinite input string, $\tau_0 \tau_1 \ldots \tau_i \ldots$, is written as $c_0 \overset{\tau_0}{\Rightarrow} c_1 \overset{\tau_1}{\Rightarrow} \ldots c_i \overset{\tau_i}{\Rightarrow} \ldots$, where $c_i \in Conf(\mathcal{P})$, $i \geq 0$. The path is also referred to as a trace of $\mathcal{P}$ if $c_0 = \langle g_0, \omega_0 \rangle$ is the initial configuration.

3.3 Büchi Pushdown System (BPDS)

A *BPDS* $\mathcal{BP}$, as defined in [1], is the Cartesian product of a BA $\mathcal{B}$ and a LPDS $\mathcal{P}$. To construct $\mathcal{BP}$, we first define (1) the input alphabet of $\mathcal{B}$ as the power set of the set of propositions that may hold on a configuration of $\mathcal{P}$ (i.e. a symbol in Σ is a set of propositions); (2) the input alphabet of $\mathcal{P}$ as the power set of the set of propositions that may hold on a state of $\mathcal{B}$ (i.e. a symbol in I is a set of propositions); and (3) two labeling functions as follows:

- $L_{\mathcal{P}2\mathcal{B}} : (G \times \Gamma) \rightarrow \Sigma$, associates the head of a LPDS configuration with the set of propositions that hold on it. Given a configuration $c \in Conf(\mathcal{P})$, we write $L_{\mathcal{P}2\mathcal{B}}(c)$ instead of $L_{\mathcal{P}2\mathcal{B}}(head(c))$ for simplicity in the rest of this paper.
- $L_{\mathcal{B}2\mathcal{P}} : Q \rightarrow I$, associates a state of $\mathcal{B}$ with the set of propositions that hold on it.

$\mathcal{BP} = ((G \times Q), \Gamma, \Delta', \langle (g_0, q_0), \omega_0 \rangle, F')$ is constructed by taking the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$. A BPDS rule $\langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle \in \Delta'$ iff $q \xrightarrow{\sigma} q' \in \delta$, $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma \rangle)$ and $\langle g, \gamma \rangle \overset{\tau}{\hookrightarrow} \langle g', w \rangle \in \Delta$, $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$. A configuration of $\mathcal{BP}$ is referred to as $\langle (g, q), \omega \rangle \in (G \times Q) \times \Gamma^*$. The set of all configurations is denoted as $Conf(\mathcal{BP})$. The labeling functions define how $\mathcal{B}$ and $\mathcal{P}$ synchronize with each other. $\langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration. $\langle (g, q), \omega \rangle \in F'$ if $q \in F$.

Given a BPDS rule $r : \langle (g, q), \gamma \rangle \hookrightarrow_{\mathcal{BP}} \langle (g', q'), \omega \rangle \in \Delta'$, for every $v \in \Gamma^*$ the configuration $\langle (g, q), \gamma v \rangle$ is an immediate predecessor of $\langle (g', q'), \omega v \rangle$, and $\langle (g', q'), \omega v \rangle$ is an immediate successor of $\langle (g, q), \gamma v \rangle$. We denote the immediate successor relation in BPDS as $\langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$, where we say this state transition follows the BPDS rule r. The reachability relation, $\Rightarrow^*_{\mathcal{BP}}$, is the reflexive and transitive closure of the immediate successor relation. A path of $\mathcal{BP}$ is a sequence of BPDS configurations, $c_0 \Rightarrow_{\mathcal{BP}} c_1 \ldots \Rightarrow_{\mathcal{BP}} c_i \Rightarrow_{\mathcal{BP}} \ldots$, where $c_i \in Conf(\mathcal{BP}), i \geq 0$. The path is also referred to as a trace of $\mathcal{BP}$ if $c_0 = \langle (g_0, q_0), \omega_0 \rangle$ is the initial configuration.

We define four concepts to assist us in analyzing the Cartesian product of $\mathcal{B}$ and $\mathcal{P}$:

Enabledness. A BPDS $\mathcal{BP}$ is constructed by synchronizing a BA $\mathcal{B}$ and a LPDS $\mathcal{P}$ through the labels on their state transitions. A Büchi transition $t_{\mathcal{B}} : q \xrightarrow{\sigma} q'$ is enabled by a LPDS configuration c (resp. a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c'$) iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c)$; otherwise $t_{\mathcal{B}}$ is disabled by c (resp. r). The LPDS rule r is enabled/disabled by the Büchi state q (resp. the Büchi transition $t_{\mathcal{B}}$) in a similar way.

Indistinguishability. Given a Büchi transition $t_{\mathcal{B}} : q \xrightarrow{\sigma} q' \in \delta$, two LPDS configurations $c, c' \in Conf(\mathcal{P})$ are (resp. a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c'$ is) indistinguishable to $t_{\mathcal{B}}$ iff $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(c) \cap L_{\mathcal{P}2\mathcal{B}}(c')$, i.e. $t_{\mathcal{B}}$ is enabled by both c and c'. On the other hand, given a LPDS rule $r : c \overset{\tau}{\hookrightarrow} c' \in \Delta$, two Büchi states $q, q' \in Q$ are (resp. a Büchi transition $t_{\mathcal{B}} : q \xrightarrow{\sigma} q'$ is) indistinguishable to r iff $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q) \cap L_{\mathcal{B}2\mathcal{P}}(q')$, i.e. r is enabled by both q and q'.

Consider a BPDS state transition, $t_{\mathcal{BP}} : \langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$ ($v \in \Gamma^*$), which is the combination of $t_{\mathcal{B}} : q \xrightarrow{\sigma} q' \in \delta$ and $t_{\mathcal{P}} : \langle g, \gamma v \rangle \overset{\tau}{\Rightarrow} \langle g', \omega v \rangle$ that follows a LPDS rule $r \in \Delta$. If the Büchi states q and q' (resp. LPDS configurations $\langle g, \gamma v \rangle$ and $\langle g', \omega v \rangle$) are both indistinguishable to r (resp. $t_{\mathcal{B}}$), $t_{\mathcal{BP}}$ can be rewritten as a BPDS path $\langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g, q'), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$ (resp. $\langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q), \omega v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$), where the concurrent state transitions of $\mathcal{B}$ and $\mathcal{P}$ are represented in an interleaved fashion with one intermediate state used.

Independence. Given a Büchi transition $t_{\mathcal{B}}$ and a LPDS rule r, if they are indistinguishable to each other, $t_{\mathcal{B}}$ and r are called independent; otherwise if either $t_{\mathcal{B}}$ or r is not indistinguishable to the other but they still enable each other, $t_{\mathcal{B}}$ and r are called dependent. The independence relation is symmetric. Furthermore, if $t_{\mathcal{B}}$ and r are dependent, (1) the BA $\mathcal{B}$ and LPDS $\mathcal{P}$ are called synchronous on them; and (2) the corresponding BPDS transitions are called synchronous transitions; otherwise if $t_{\mathcal{B}}$ and r are independent, (1) $\mathcal{B}$ and $\mathcal{P}$ are called asynchronous on them; and (2) the corresponding BPDS transitions are called asynchronous transitions.

Commutativity. Without affecting the reachability property, if a BPDS state transition, $t_{\mathcal{BP}} : \langle (g, q), \gamma v \rangle \Rightarrow_{\mathcal{BP}} \langle (g', q'), \omega v \rangle$ can be rewritten respectively as two BPDS paths

such that $\langle (g,q),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g,q'),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q'),\omega v\rangle$ and $\langle (g,q),\gamma v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q),\omega v\rangle \Rightarrow_{\mathcal{BP}} \langle (g',q'),\omega v\rangle$, the corresponding Büchi transition $t_{\mathcal{B}}$ and LPDS rule r are called commutative. By definition, commutativity is equivalent to independence but seen under a different light, which will help the presentation of the paper.

3.4 Static Partial Order Reduction

One common method for reducing the complexity of model checking asynchronous systems is partial order reduction [10], which is based on the observation that properties often do not distinguish among the state transition orders. Traditional partial order reduction algorithms use an explicit state representation and depth first search, where both the states and transitions to be explored are selected during the model checking process. Kurshan et al. [3] developed an alternative approach called static partial order reduction, where the key idea is to apply partial order reduction when a model is generated from the system specification. Thus, no modification to the model checker is necessary. The model is reduced during the compilation phase by exploring the structure of the system specification. Any model checker that accepts this kind of model can then be applied to solve the verification problem.

4 Reachability Analysis of BPDS

4.1 Reachability Analysis of BPDS without Reduction

For reachability analysis, we have demonstrated [1] that a BPDS $\mathcal{BP}$ can be converted into a PDS $\mathcal{P}'$, which we refer to as the verification model, so that model checkers for PDS (or PDS-equivalent models) can be readily utilized. It is important to note that $\mathcal{P}'$ is a standard PDS in the sense that $\mathcal{P}'$ does not have inputs. Given $\mathcal{BP} = ((G\times Q),\Gamma,\Delta',\langle (g_0,q_0),\omega_0\rangle,F')$, we construct $\mathcal{P}' = (G_{\mathcal{P}'},\Gamma_{\mathcal{P}'},\Delta_{\mathcal{P}'},c_0)$ such that $G_{\mathcal{P}'} = (G\times Q)$, $\Gamma_{\mathcal{P}'} = \Gamma$, $c_0 = \langle (g_0,q_0),\omega_0\rangle$, and $\Delta_{\mathcal{P}'}$ is converted from $\Delta' = \delta\times\Delta$, where $\forall t = q \xrightarrow{\sigma} q' \in \delta$ and $\forall r = \langle g,\gamma\rangle \overset{\tau}{\hookrightarrow} \langle g',\omega\rangle \in \Delta$, if t and r are dependent, add a rule $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q'),\omega\rangle$ to $\Delta_{\mathcal{P}'}$, i.e. $\mathcal{B}$ and $\mathcal{P}$ must transition synchronously; else if t and r are independent, add three rules to $\Delta_{\mathcal{P}'}$: (1) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g,q'),\gamma\rangle$, i.e. $\mathcal{B}$ transitions and $\mathcal{P}$ loops; (2) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q),\omega\rangle$, i.e. $\mathcal{P}$ transitions and $\mathcal{B}$ loops; and (3) $\langle (g,q),\gamma\rangle \hookrightarrow \langle (g',q'),\omega\rangle$, i.e. $\mathcal{B}$ and $\mathcal{P}$ transition together. Rules (1) and (2) represent the non-deterministic delays that may occur between $\mathcal{B}$ and $\mathcal{P}$. Non-deterministic delays do not affect reachability properties. Rule (3) can be represented by Rules (1) and (2) together because $\mathcal{B}$ and $\mathcal{P}$ are asynchronous; however we include Rule (3) here to help the presentation of Section 4.2. The correctness of the conversion that $\mathcal{P}'$ preserves the reachability property of $\mathcal{BP}$ is proved in [1].

4.2 Efficient Reachability Analysis Based on Static Partial Order Reduction

As discussed above, when a BPDS $\mathcal{BP}$ is converted to a PDS $\mathcal{P}'$ by the naïve approach, both the size of the state space and the number of the transition rules remain the same. For example, the set of transition rules is the product of δ that belongs to $\mathcal{B}$ and Δ that

belongs to $\mathcal{P}$. However, a complete product is not necessary when $\mathcal{B}$ and $\mathcal{P}$ are asynchronous. Without affecting the verification result, static partial order reduction can be applied to reduce the transition rules generated by the product. The reduced PDS model $\mathcal{P}'_r$ will have a smaller set of transition rules $\Delta_{\mathcal{P}'_r} \subseteq \Delta_{\mathcal{P}'}$ and fewer state transition traces while still preserving the reachability properties of $\mathcal{P}'$. Figure 1 illustrates the verification process that supports the reduction.

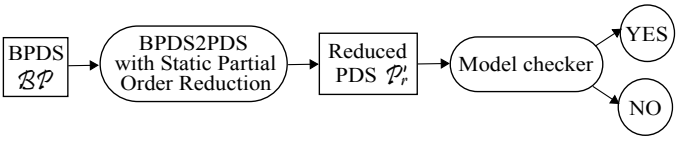
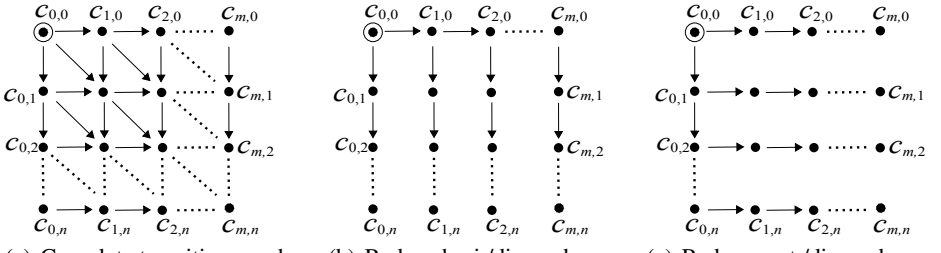


Fig. 1. Reachability analysis of BPDS with static partial order reduction

Our reduction is based on the observation that when $\mathcal{B}$ and $\mathcal{P}$ transition asynchronously, one can run continuously while the other one loops. Figure 2 illustrates the idea of reducing a BPDS state transition graph that starts from the configuration $c_{0,0}$. Figure 2(a)



(a) Complete transition graph (b) Reduce hori./diag. edges (c) Reduce vert./diag. edges

Fig. 2. Reducing state transition edges without affecting the reachability from $c_{0,0}$ when BA and LPDS are asynchronous

is a complete state transition graph. There are three types of transition edges: (1) a horizontal edge represents a transition when $\mathcal{B}$ transitions and $\mathcal{P}$ loops, which follows a BPDS rule in the form of $\langle (g,q),\gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g,q'),\gamma\rangle$; (2) a vertical edge represents a transition when $\mathcal{P}$ transitions and $\mathcal{B}$ loops, which follows a BPDS rule in the form of $\langle (g,q),\gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g',q),w\rangle$; and (3) a diagonal edge represents a transition when $\mathcal{B}$ and $\mathcal{P}$ transition together, which follows a BPDS rule in the form of $\langle (g,q),\gamma\rangle \hookrightarrow_{\mathcal{BP}} \langle (g',q'),w\rangle$. For every configuration $c_{i,j} = \langle (g,q),\gamma v\rangle$ ($0 \le i \le m$ and $0 \le j \le n$) as well as the Büchi transition $t_{\mathcal{B}} : q \xrightarrow{\sigma} q'$ and the LPDS rule $r : \langle g,\gamma\rangle \overset{\tau}{\hookrightarrow} \langle g',\omega\rangle$ that are both enabled on $c_{i,j}$, if $t_{\mathcal{B}}$ and r are independent, we can reduce many state transitions in Figure 2(a) without affecting the reachability from $c_{0,0}$ to other configurations in the graph. Figure 2(b) and Figure 2(c) illustrate two reductions that reduce horizontal/diagonal transition edges and vertical/diagonal transition edges respectively. This kind of reduction can significantly reduce the transition rules of $\mathcal{BP}$, where Büchi transitions and LPDS rules are independent.

Now we present an optimization of our previous approach, where the reduction is applied during the rule generation phase of constructing the verification model $\mathcal{P}'_r$. We define a set of heads, $SensitiveSet$, on $Conf(\mathcal{P})$ as follows:

Definition 1. $SensitiveSet = \{\ head(\langle g_0, \omega_0\rangle)\ \} \bigcup \{\ head(c') \mid \exists r = c \overset{\tau}{\hookrightarrow} c' \in \Delta,$ $\exists t_{\mathcal{B}} \in \delta$, *r and* $t_{\mathcal{B}}$ *are dependent* $\}$, *where* $\langle g_0, \omega_0\rangle$ *is the initial configuration of* $\mathcal{P}$.

The concept of $SensitiveSet$ is similar to that of sleep set [10]. However, instead of identifying transitions that are not necessary to be executed (i.e. reducible) at a state, $SensitiveSet$ identifies transitions that should be kept (i.e. irreducible). Algorithm 1 applies the reduction following the idea illustrated in Figure 2(b), where the horizontal/diagonal edges are reduced. If the LPDS rule r and the Büchi transition $t_{\mathcal{B}}$ are

Algorithm 1. BPDS2PDS_SPOR($\delta \times \Delta$)

1: $\Delta_{sync} \leftarrow \emptyset$, $\Delta_{vert} \leftarrow \emptyset$, $\Delta_{hori} \leftarrow \emptyset$
2: **for all** $r : \langle g, \gamma\rangle \overset{\tau}{\hookrightarrow} \langle g', \omega\rangle \in \Delta$ **do**
3: **for all** $t_{\mathcal{B}} : q \overset{\sigma}{\rightarrow} q' \in \delta$ **and** $\sigma \subseteq L_{\mathcal{P}2\mathcal{B}}(\langle g, \gamma\rangle)$ **and** $\tau \subseteq L_{\mathcal{B}2\mathcal{P}}(q)$ **do**
4: **if** r and $t_{\mathcal{B}}$ are dependent **then**
5: {*When* $\mathcal{B}$ *and* $\mathcal{P}$ *are synchronous on* r *and* $t_{\mathcal{B}}$}
6: $\Delta_{sync} \leftarrow \Delta_{sync} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q'), \omega\rangle\}$
7: **else**
8: {*For vertical edges (see Figure 2(b)), when* $\mathcal{P}$ *transitions and* $\mathcal{B}$ *loops*}
9: $\Delta_{vert} \leftarrow \Delta_{vert} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g', q), \omega\rangle\}$
10: **if** $\langle g, \gamma\rangle \in SensitiveSet$ **then**
11: {*For horizontal edges (see Figure 2(b)), when* $\mathcal{B}$ *transitions and* $\mathcal{P}$ *loops*}
12: $\Delta_{hori} \leftarrow \Delta_{hori} \bigcup \{\langle (g, q), \gamma\rangle \hookrightarrow \langle (g, q'), \gamma\rangle\}$
13: **end if**
14: **end if**
15: **end for**
16: **end for**
17: $\Delta_{\mathcal{P}'_r} \leftarrow \Delta_{sync} \bigcup \Delta_{vert} \bigcup \Delta_{hori}$
18: **return** $\Delta_{\mathcal{P}'_r}$

dependent, $\mathcal{B}$ and $\mathcal{P}$ must transition synchronously as the set of rules, Δ_{sync}, generated in line 6; otherwise, asynchronous transitions are generated. The set of rules, Δ_{vert}, generated in line 9 represent the vertical edges, i.e. when $\mathcal{P}$ transitions and $\mathcal{B}$ loops. The set of rules, Δ_{hori}, representing the horizontal edges, i.e. when $\mathcal{B}$ transitions and $\mathcal{P}$ loops, are generated in line 12 only if $head(r)$ belongs to $SensitiveSet$.

In Algorithm 1, a diagonal rule is reduced if $\mathcal{B}$ and $\mathcal{P}$ are asynchronous on the corresponding Büchi transition and LPDS rule. This kind of reduction does not affect any reachability property, because the diagonal rule can be represented by one horizontal rule and one vertical rule respectively. A horizontal rule is reduced if the head of the corresponding LPDS rule in $\mathcal{P}$ does not belong to $SensitiveSet$. There is a special set of heads, $DivideSet = \{\ h \mid h \in SensitiveSet, \forall r = c \overset{\tau}{\hookrightarrow} c' \in \Delta$ and $\forall t_{\mathcal{B}} \in \delta$, if $head(c) = h$ then r and $t_{\mathcal{B}}$ are not dependent $\}$. Informally, $DivideSet$ describes a set of configurations that can be considered as divide-lines (in the traces of $\mathcal{P}$

projected from the traces of $\mathcal{BP}$) for two adjacent LPDS transitions that are respectively synchronous and asynchronous with the state transitions of $\mathcal{B}$. Given a trace of $\mathcal{P}'_r$ in the form of $\langle(g_0, q_0), \omega_0\rangle \Rightarrow \ldots \Rightarrow \langle(g_j, q_j), \omega_j\rangle \Rightarrow \ldots \Rightarrow \langle(g_k, q_k), \omega_k\rangle \Rightarrow \ldots (0 \leq j < k)$, if $head(\langle g_j, \omega_j\rangle) \in DivideSet$ and $\langle(g_k, q_k), \omega_k\rangle$ is the first configuration satisfying $head(\langle g_k, \omega_k\rangle) \in SensitiveSet$ after $\langle(g_j, q_j), \omega_j\rangle$, we can infer that no horizontal transition occurs between $\langle(g_{j+1}, q_{j+1}), \omega_{j+1}\rangle$ and $\langle(g_k, q_k), \omega_k\rangle$ in the trace (i.e. $q_{j+1} = q_k$), because the horizontal transitions have been reduced.

Theorem 1. *$\mathcal{P}'_r$ preserves the reachability of $\mathcal{P}'$ from the initial configuration.*

Proof. It is easy to observe that $\mathcal{P}'_r$ and $\mathcal{P}'$ have the same state space and initial configuration, so the question is to prove that (1) given a configuration c and a trace of $\mathcal{P}'$ in the form of $T : c_0 \Rightarrow c_1 \ldots \Rightarrow c_i \Rightarrow c$, there is a corresponding trace of $\mathcal{P}'_r$ such that $T' : c_0 \Rightarrow c'_1 \ldots \Rightarrow c'_j \Rightarrow c$; and (2) vice versa.

Two types of transitions are reduced in $\mathcal{P}'_r$, compared to $\mathcal{P}'$. As explained above, the reduction of diagonal transitions does not affect any reachability property. We prove that the reduction of horizontal transitions does not affect the correctness of (1) by induction. If $|T| = 0$, i.e. $c = c_0$, the reachability trivially holds on $\mathcal{P}'_r$. If $|T| = 1$, because there is no horizontal transition reduced on the initial configuration, for any transition $c_0 \Rightarrow c$ of $\mathcal{P}'$ there must be a corresponding trace of $\mathcal{P}'_r$ that preserves the reachability. Given a trace $T : c_0 \Rightarrow c_1 \ldots \Rightarrow c_i \Rightarrow c'$ $(i \geq 0)$ of $\mathcal{P}'$ where $|T| = i+1$, if there exists a trace $T' : c_0 \Rightarrow c'_1 \ldots \Rightarrow c'_j \Rightarrow c'$ $(j \geq 0)$ of $\mathcal{P}'_r$ where $|T'| = j + 1$, we show that for all $c \in Conf(\mathcal{P}')$ and $t_{\mathcal{P}'} : c' \Rightarrow c$ of $\mathcal{P}'$, there is a trace of $\mathcal{P}'_r$ such that $c_0 \Rightarrow^* c$. Recall that the horizontal transitions are reduced in $\mathcal{P}'_r$ except at configurations whose heads belong to $SensitiveSet$, so we need to prove that this reduction does not affect the reachability if $t_{\mathcal{P}'}$ involves a horizontal transition that is reduced in $\mathcal{P}'_r$. In the trace T', we can always find a configuration $c'_k = \langle(g_k, q_k), \omega_k\rangle$ $(0 \leq k \leq j)$ such that c'_k is the last configuration satisfying $head(\langle g_k, \omega_k\rangle) \in SensitiveSet$. Thus, the path from c'_k to c' has the form of $(c'_k : \langle(g_k, q_k), \omega_k\rangle) \Rightarrow \langle(g_{k+1}, q_k), \omega_{k+1}\rangle \Rightarrow \ldots \Rightarrow (c' : \langle(g_{j+1}, q_k), \omega_{j+1}\rangle)$, where $\mathcal{B}$ always loops at the state q_k after c'_k. Because the horizontal transitions are reduced on the configurations after c'_k, $\mathcal{P}'_r$ cannot directly have the transition $(c' : \langle(g_{j+1}, q_k), \omega_{j+1}\rangle) \Rightarrow (c : \langle(g_{j+1}, q_{k+1}), \omega_{j+1}\rangle)$, i.e. the corresponding BPDS rule $\langle(g_{j+1}, q_k), \gamma_{j+1}\rangle) \hookrightarrow_{\mathcal{BP}} \langle(g_{j+1}, q_{k+1}), \gamma_{j+1}\rangle$ (γ_{j+1} is the top stack symbol of ω_{j+1}) does not exist after the reduction. According to the commutativity between independent Büchi transitions and LPDS rules, we can shift this transition backward to the position right after c'_k where the horizontal transitions are not reduced. In this case, the path is $\underline{(c'_k : \langle(g_k, q_k), \omega_k\rangle) \Rightarrow \langle(g_k, q_{k+1}), \omega_k\rangle} \Rightarrow \langle(g_{k+1}, q_{k+1}), \omega_{k+1}\rangle \Rightarrow \ldots \Rightarrow (c : \langle(g_{j+1}, q_{k+1}), \omega_{j+1}\rangle)$, so we proved that there is a trace $c_0 \Rightarrow^* c$ of $\mathcal{P}'_r$.

On the other direction, (2) always holds because $\Delta_{\mathcal{P}'_r} \subseteq \Delta_{\mathcal{P}'}$. For every rule of $\mathcal{P}'_r$, $\mathcal{P}'$ has the same rule. Thus for every trace of $\mathcal{P}'_r$, $\mathcal{P}'$ has the same trace. □

Complexity analysis. Let n_{SR} be the number of LPDS rules (in Δ) whose heads belong to $SensitiveSet$, and n_{sync} be the number of PDS rules (in $\Delta_{\mathcal{P}'_r}$) where $\mathcal{B}$ and $\mathcal{P}$ transition synchronously on the corresponding Büchi transitions and LPDS rules. We have $|\Delta_{hori}| = n_{SR} \times |\delta|$ and $|\Delta_{sync}| = n_{sync}$. As illustrated in Figure 2, asynchronous transitions can be organized as triples where each one includes a vertical transition, a horizontal transition, and a diagonal transition, so we have $|\Delta_{vert}| = \frac{|\delta \times \Delta| - n_{sync}}{3}$.

The number of rules generated in Algorithm 1 is $|\Delta_{\mathcal{P}'_r}| = n_{sync} + \frac{|\delta \times \Delta| - n_{sync}}{3} + n_{SR} \times |\delta| = \frac{2}{3}n_{sync} + \frac{|\delta \times \Delta|}{3} + n_{SR} \times |\delta|$. The size of transition rules reduced is $|\Delta'| - |\Delta_{\mathcal{P}'_r}| = \frac{2}{3}|\delta \times \Delta| - \frac{2}{3}n_{sync} - n_{SR} \times |\delta|$. We can infer from this expression that the fewer places that $\mathcal{B}$ and $\mathcal{P}$ transition synchronously the more transition rules Algorithm 1 can reduce.

Discussions. Algorithm 1 makes a product of the transition relations respectively from the BA and LPDS, where all the transition rules are explored. Obviously, this process could be inefficient if the BA and LPDS are represented in a flattened approach, since the sizes of the transition relations can be exponentially large. Symbolic representations are efficient to model transition relations; therefore the cost of Algorithm 1 can be exponentially smaller on symbolic representations than that on flattened representations. However, the symbolic rules should be properly separated for the reduction to be effective. For example, if there is only one giant symbolic transition rule for each transition relation, Algorithm 1 will have no reduction effect. Symbolic rules are commonly differentiated by their control locations. This explains why the idea in Figure 2(b) is used instead of that in Figure 2(c), because LPDS usually has a better control-flow structure than BA.

5 Implementation

We apply the BPDS model in the verification of Windows device drivers with their formal hardware interface models as illustrated in Figure 3, where software is represented as LPDS and hardware is represented as BA. From the view of software, we

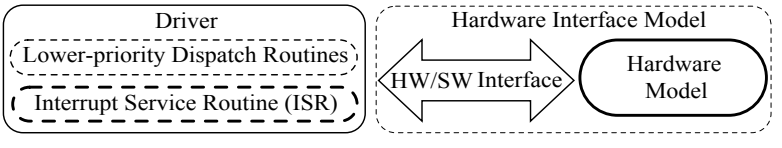


Fig. 3. Driver-centric co-verification

specify both the HW/SW interface and the hardware model, which together we refer to as a hardware interface model. The HW/SW interface describes how hardware and software should transition synchronously when they interact through their interfaces. The hardware model describes the desired hardware behaviors when hardware and software transition asynchronously, i.e. when there is no HW/SW interaction.

First, we present several preliminary definitions for our implementation. Second, we elaborate on the specification of the HW/SW interface and the hardware model respectively by examples. Third, we illustrate our automatic co-verification tool, CoVer.

5.1 Preliminary Definitions

We use Transaction Level Modeling (TLM) to specify the hardware interface model. TLM is a commonly used approach to hardware system-level specification, and we have

designed a specification language, modelC, for our TLM specification. The modelC language uses C semantics with two extensions to support non-determinism and relative atomicity (see definitions below). In modelC, (1) we treat numbers as bounded integers, so hardware registers can be properly modeled; and (2) the global hardware state space is static, i.e. there is no dynamic memory allocation.

Hardware transaction. In co-verification, the interaction between hardware and software is relevant rather than the implementation details of a hardware device; therefore it is unnecessary to preserve the clock-driven semantic feature. We define a hardware transaction to represent a hardware state transition in an arbitrarily long but finite sequence of clock cycles. Hardware transactions are atomic to software. The concept of hardware transaction preserves hardware design logic that is visible to software, but hides details only necessary for synthesizable Register Transfer Level (RTL) design.

Hardware transaction function. We define a transaction function as a C function that describes a set of hardware transactions (i.e. state transitions). Because transactions are atomic, the intermediate states of hardware during a transaction is not visible to software. We define the current-states and next-states of a transaction function respectively as $\rho \subseteq Q$ representing the hardware states when entering the function and $\rho' \subseteq Q$ representing the hardware states when exiting the function. Formally, a transaction function $\mathcal{F} : Q \times Q$ describes a set of state transitions. Following this definition, any terminating C function can be treated as a transaction function.

Relative atomicity. Relative atomicity has two key ideas: (1) hardware transactions are atomic from the view of software; and (2) Interrupt Service Routines (ISRs) are atomic to other lower-priority software routines. In device/driver applications, when hardware fires an interrupt, the Operating System (OS) calls the ISRs that are registered in the interrupt vector table sequentially until an ISR acknowledges its ownership of the interrupt. During this process, only one ISR can run at a time and other hardware interrupts are suppressed [12]. The interrupted thread can continue its execution only after the interrupting ISR terminates.

Software synchronization points. As the concrete counterpart of the $SensitiveSet$ concept, we define software synchronization points as a set of program locations[1] where the program statements right before these locations may be dependent with some of the hardware state transitions. In general, there are three types of software synchronization points: (1) the point where the program is initialized; (2) those points right after software reads/writes hardware interface registers; and (3) those points where hardware interrupts may affect the verification results. The first and second types are straightforward for hardware and software to transition synchronously. We may understand the third type in such a way that the effect of interrupts (by executing ISRs) may influence certain program statements, e.g. the statements that access global variables.

[1] Assuming the program is preprocessed to ensure that every statement is atomic from the view of hardware.

5.2 Specifying Hardware Interface Model

In the specification of the hardware BA model, $\mathcal{B} = (\Sigma, Q, \delta, q_0, F)$, the alphabet Σ is the power set of the set of propositions induced by software interface events (see definition below); the set of states Q is defined by global variables; the initial state q_0 is given by an initialization function; and the transition relation $R = R_{evt} \cup R_{model}$ has two parts: R_{evt}, is a set of transitions that are dependent with at least one of the software LPDS transition rules; R_{model}, is a set of transitions that are not dependent with any of the LPDS transition rules. Informally, R_{evt} is described by the HW/SW interface and R_{model} is described by the hardware model. In this paper, we are interested only in safety properties; therefore the Büchi constraint F is not necessary to be specified.

Specifying the HW/SW interface. The HW/SW interface, as the abstraction of the HW/SW layers between the target device and driver, propagates the hardware (resp. software) interface events to software (resp. hardware).

Figure 4 illustrates an example of a software interface event function in response to a register write operation. The keyword `__atomic` indicates that `WritePortA` is a transaction function atomic from the view of software. This transaction function describes a set of state transitions, $R'_{evt} \subseteq R_{evt}$, when the driver writes to the interface register, PortA, of the Sealevel PIO-24 digital I/O device (see Section 6). Figure 5

```
__atomic VOID WritePortA(UCHAR ucRegData) {
  // If Port A is configured as an "input" port
  if ( g_DIORegs.CW.CWD4 == 1 ) {
    // Write to the output register instead of the port
    g_DIOState.OutputRegA.ucValue = ucRegData;

  } else { // Otherwise, configured as an "output" port
    // Update both the port and the output register
    g_DIORegs.A.ucValue = ucRegData;
    g_DIOState.OutputRegA.ucValue = ucRegData;
  }
}
```

Fig. 4. An implementation of a software interface event

```
VOID WRITE_REGISTER_UCHAR
    (PUCHAR Register, UCHAR ucData) {
  switch ( Register ) {
    case REG_PORTA:  WritePortA(ucData); return;
    case REG_PORTB:  WritePortB(ucData); return;
    ...
    case REG_CONFIG:  WriteConfig(ucData); return;
    case REG_STATUS:  WriteStatus(ucData); return;
    default: abort "Register address error."; return;
  }
}
```

Fig. 5. Relating register calls to software interface events

illustrates an example how function calls to a software write-register function (originally provided by the OS) are related to interface event functions. A software interface event happens when the entry stack symbol of the interface event function is reached.

When hardware fires an interrupt, the ISR should be invoked to service this interrupt. The HW/SW interface simulates this process as shown in Figure 6. The variable `IsrRunning` represents the software status and the variable `InterruptPending` represents the hardware status. The function `RunIsr` has three parts, (1) check/prepare the precondition before invoking the ISR; (2) invoke the ISR; and (3) set both the hardware and software to proper status after ISR. The first and third parts describe synchronous state transitions of both hardware and software. Formally, when hardware (the BA) fires an interrupt, i.e. the interrupt pending status is set to be true, the corresponding state transitions in software (the LPDS) will be enabled, so the BA and the LPDS will transition synchronously in the next state transition.

```
VOID RunIsr() {
  _atomic {
    // Make sure only one ISR is invoked
    if ( (IsrRunning == TRUE) ||
         (InterruptPending == FALSE) )
      return;
    IsrRunning = TRUE;
  }

  // Invoke the ISR
  IsrFoo();

  _atomic {
    IsrRunning = FALSE;
    InterruptPending = FALSE;
  }
}
```

Fig. 6. Interrupt monitoring function

```
_atomic VOID Run_DIO() {

  // non-deterministic choices
  switch ( choice() ) {

    // Port I/O Management
    case 0: RunPorts(); break;

    // Interrupt Management
    case 1: RunInterrupt(); break;
    ...
}
```

Fig. 7. The transaction function of the Sealevel PIO-24 card

```
VOID HWInstr() {

  // non-deterministic choices
  while( choice() ) {

    // Run hardware transaction
    Run_DIO();

    // If interrupt has been fired
    RunIsr();
  }
}
```

Fig. 8. The hardware instrumentation function

Specifying the hardware model. The hardware model describes the desired hardware behaviors when hardware works asynchronously with software to realize system functionalities. Conceptually, the behavior of the hardware model is represented as a set of state transitions, R_{model}, where all the transitions are labeled by a set of propositions that hold when no software interface event happens. Figure 7 illustrates an example of a transaction function, `Run_DIO`, that specifies the set of state transitions, R_{model}, for the digital I/O device. When `Run_DIO` is executed multiple times, the stub-functions such as `RunPorts` and `RunInterrupt` are non-deterministically invoked to simulate the concurrent sub-modules of the hardware device.

Hardware instrumentation function. We define a C function to invoke independent hardware transaction functions (for the hardware model) and ISRs. Figure 8 illustrates such an example, where `RunIsr` is invoked right after every hardware transaction, `Run_DIO`. If an interrupt is fired due to a hardware state transition by executing `Run_DIO`, the context-switch to the ISR is modeled as a function call, where the execution privilege switches back to the interrupted thread only after the ISR returns. This approach is correct to simulate the context-switches because ISRs are relatively atomic to other driver routines. The non-deterministic while-loop simulates the delays of either software or hardware. This is correct when only safety properties are verified.

5.3 Co-verification Tool, CoVer

Our co-verification tool, CoVer, has two automatic steps. First, the frontend instruments (i.e. make the product of) the driver with the hardware interface model to generate a C program, which conceptually is the reduced verification model $\mathcal{P}'_r$ discussed in Section 4.2. Second, the SLAM engine checks the reachability property (in the form of a SLIC rule [4]) of the C program.

The instrumentation step has two parts. First, the dependent HW/SW transitions when driver writes hardware registers are modeled by replacing the implementation of the driver programming interfaces (see Figure 5), which is provided in the harness

of Static Driver Verifier [4]. Second, CoVer inserts function calls to the hardware instrumentation function `HWInstr` into the C code of the driver, between the driver statements. Without reductions, the function calls need to be inserted after every driver statement. Using our reduction algorithm, CoVer first detects the software synchronization points in the driver code and then inserts the function calls only at those detected points. Conceptually, the instrumentation lets hardware run continuously for all the possibilities after every HW/SW synchronous transition. Compared to the trivial approach that inserts `HWInstr` after every software statement to simulate the HW/SW concurrent state transitions, our approach can significantly reduce the complexity of the verification model, because the number of software synchronization points are usually very small in common applications.

6 Evaluation

We have applied our approach to the verification of two fully functional Windows device drivers: (1) the Sealevel PCI (Peripheral Component Interconnect) PIO-24 Digital I/O card driver from Open Systems Resources (OSR), and (2) the Intel 82557/82558 based PCI Ethernet adapter driver from Microsoft Windows Driver Kit (WDK) samples. We developed hardware interface models respectively for the drivers and verified two kinds of properties: (1) whether a driver callback function[2] accesses the hardware interface registers in correct ways, e.g. a command should not be issued when the hardware is busy; and (2) whether a driver callback function can cause an out-of-synchronization between the driver and the device. In other words, we check if the return value of a driver callback function correctly indicates the current hardware state. Because both of the drivers have been provided to public as samples for years, we did not expect to find many bugs. However, CoVer detected seven real bugs. All these bugs can cause malfunction of the devices/drivers, where the symptoms include data loss, interrupt storm, device hang, etc. Our evaluation runs on a Lenovo ThinkPad notebook with Dual Core 2.66GHz CPU and 4GB memory. We set the timeout and spaceout threshold as 3000 seconds and 2000MB respectively.

Table 1 presents the statistics on the verification of the PIO-24 driver with its hardware interface model. CoVer detected four bugs and proved two properties of the driver. For example, the driver has two global variables to maintain the I/O request status and the device I/O port status respectively. The values of the two variables become inconsistent when the ISR interrupts the callback function `EvtDeviceControl` at a specific program location. This inconsistency will cause the driver to return invalid data to user applications later, which violates the rule `InvalidRead`. Another serious bug (detected by the rule `ProperISR1`) of this driver can cause interrupt storm. The design of the device expects interrupts being repeatedly generated in certain configuration, however the driver does not handle the interrupts correctly which will cause interrupts being fired more frequently than that can be consumed, i.e. interrupt storm. As a comparison, the Ethernet adapter driver disables the interrupt first and re-enables it after the

[2] Windows OS invokes the predefined driver callback functions to service the I/O requests from user applications.

Table 1. Statistics on the co-verification of the Sealevel PIO-24 device/driver

Size of the driver (# of lines)						1724
Size of the hardware interface model (# of lines)						1232
Rule	Description	No Reduction		Reduction		Result
		Time (Sec)	Mem. (MB)	Time (Sec)	Mem. (MB)	
DevD0Entry	Driver and device will not go out-of-synchronization when starting.	391.3	293	214.3	181	Passed
DevD0Exit	Driver and device will not go out-of-synchronization when stopping.	71.1	69	38.4	43	Passed
IsrCallDpc	ISR will not queue DPC without reading specific hardware registers.	Timeout	N/A	700.5	218	Failed
InvalidRead	Driver will not read any invalid input data.	589.4	132	91.3	66	Failed
ProperISR1	ISR will clear the device interrupt-pending status before return.	58.9	58	35.2	43	Failed
ProperISR2	ISR will not acknowledge the interrupt fired by other devices.	74.1	62	28.7	37	Failed

interrupt processing is completed later in DPC (Deferred Procedure Call). This prevents the situation when interrupts overwhelm the PCI bus.

Table 2 presents the statistics on the verification of the Intel 82557/82558 based PCI Ethernet adapter driver with its hardware interface model. CoVer detected three bugs and proved five properties of the driver. For example, CoVer detects a bug that violates the rule `DevD0Entry` and reports an error trace where the callback function `EvtDeviceD0Entry` returns `TRUE` even if the driver fails to initialize the device correctly. This is a direct violation of Windows device driver programming standards and will cause the device unusable without the OS being notified. The error trace also illustrates that the driver continues its attempts to initialize the device even after the previous device operations have failed. This may cause the device permanently unaccessible. Compared to the PIO-24 device/driver, the Ethernet adapter device/driver have more comprehensive functionalities and implementation, where the static partial order reduction is clearly necessary for most of the rules to be even verified.

Table 2. Statistics on the co-verification of the Intel PCI Ethernet adapter device/driver

Size of the driver (# of lines)						14406
Size of the hardware interface model (# of lines)						3518
Rule	Description	No Reduction		Reduction		Result
		Time (Sec)	Mem. (MB)	Time (Sec)	Mem. (MB)	
DevD0Entry	Driver and device will not go out-of-synchronization when starting.	1328.3	758	367.1	182	Failed
DevD0Exit	Driver and device will not go out-of-synchronization when stopping.	Timeout	N/A	206.6	143	Failed
IsrCallDpc	ISR will not queue DPC without reading specific hardware registers.	64.1	99	39.9	79	Passed
ProperISR1	ISR will clear the device interrupt-pending status before return.	48.9	59	32.6	52	Passed
ProperISR2	ISR will not acknowledge the interrupt fired by other devices.	779.3	291	407.4	199	Passed
DoubleCUC	Driver will not issue a command while the command unit is busy.	Timeout	N/A	602.4	238	Failed
DoubleRUC	Driver will not issue a command while the receiving unit is busy.	N/A	Spaceout	1797.3	231	Passed
ProperReset	Driver uses a correct sequence to reset the device.	Timeout	N/A	86.9	71	Passed

7 Conclusion and Future Work

We have presented an efficient approach to reachability analysis of BPDS models for HW/SW co-verification. The key idea of this approach is to reduce unnecessary state transition orders between hardware and software, so there are fewer possibilities to be

explored in verification. We have implemented this approach in our co-verification tool, CoVer, and successfully applied it to co-verify two Windows device drivers with their device models. CoVer proved seven properties and detected seven previously undiscovered software bugs which can cause serious system failures. Evaluation shows that the reduction can significantly scale co-verification. These results demonstrate that our approach is very promising in ensuring the correct interactions between hardware and software. For the next step, we plan to apply our approach to more devices and drivers.

Acknowledgement. This research received financial support from National Science Foundation of the United States (Grant #: 0916968). We thank Con McGarvey, Onur Ozyer, and Peter Wieland for evaluating our findings of device driver bugs.

References

1. Li, J., Xie, F., Ball, T., Levin, V., McGarvey, C.: An automata-theoretic approach to hardware/software co-verification. In: Proc. of FASE (2010)
2. Schwoon, S.: Model-Checking Pushdown Systems. PhD thesis (2002)
3. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Static partial order reduction. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, p. 345. Springer, Heidelberg (1998)
4. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: Proc. of EuroSys (2006)
5. Kurshan, R.P., Levin, V., Minea, M., Peled, D., Yenigün, H.: Combining software and hardware verification techniques. FMSD (2002)
6. Xie, F., Yang, G., Song, X.: Component-based hardware/software co-verification for building trustworthy embedded systems. JSS 80(5) (2007)
7. Monniaux, D.: Verification of device drivers and intelligent controllers: a case study. In: Proc. of EMSOFT (2007)
8. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM Trans. Program. Lang. Syst. (2000)
9. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Application to model-checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, Springer, Heidelberg (1997)
10. Godefroid, P.: Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem. PhD thesis (1994)
11. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach. Princeton University Press, Princeton (1994)
12. Microsoft: Synchronizing interrupt code. In: MSDN (2009), msdn.microsoft.com/en-us/library/aa490313.aspx

LTSMIN: Distributed and Symbolic Reachability

Stefan Blom*, Jaco van de Pol, and Michael Weber

Formal Methods and Tools, University of Twente, The Netherlands
{sccblom,vdpol,michaelw}@cs.utwente.nl

In model checking, analysis algorithms are applied to large graphs (state spaces), which model the behavior of (computer) systems. These models are typically generated from specifications in high-level languages. The LTSMIN toolset[1] provides means to generate state spaces from high-level specifications, to check safety properties on-the-fly, to store the resulting labelled transition systems (LTSs) in compressed format, and to minimize them with respect to (branching) bisimulation.

1 Motivation: A Modular, High-Performance Model Checker

The LTSMIN toolset provides a new level of modular design to high-performance model checkers. Its distinguishing feature is the wide spectrum of supported specification languages and model checking paradigms. On the language side (Sec. 3.1), it supports process algebras (MCRL), state based languages (PROMELA, DVE) and even discrete abstractions of ODE models (MAPLE, GNA). On the algorithmic side (Sec. 3.2), it supports two main streams in high-performance model checking: reachability analysis based on BDDs (symbolic) and on a cluster of workstations (distributed, enumerative). LTSMIN also incorporates a distributed implementation of state space minimization, preserving strong or branching bisimulation.

For end users, this implies that they can exploit other, scalable, verification algorithms than supported by their native tools, without changing specification language. Our experiments (Sec. 4) show that the LTSMIN toolset can match, and often outperform, existing tools tailored to their own specification language.

From an algorithm engineering point of view, LTSMIN fosters the availability of benchmark suites across multiple specification languages and verification communities. This makes benchmarking studies more robust, by separating out language-specific issues, which is of separate scientific interest. The LTSMIN toolset integrates very well with existing third-party tools (Sec. 3.3), for the benefit of their users, and also for the independent certification of model checking results.

The technical enabler of the LTSMIN toolset is its PINS interface (Sec. 2). This general abstraction of specification languages places very few constraints on their features, evident by the variety of supported languages (Sec. 3.1) and algorithms. PINS still enables the algorithms to exploit the parallel structure inherent in many specifications. Several optimizations are implemented as generic PINS2PINS wrappers, abstracting

* This research has been partially funded by the EC project EC-MOAN (FP6-NEST 043235).

[1] http://fmt.cs.utwente.nl/tools/ltsmin/, current version: 1.5, available as open-source software.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 354–359, 2010.

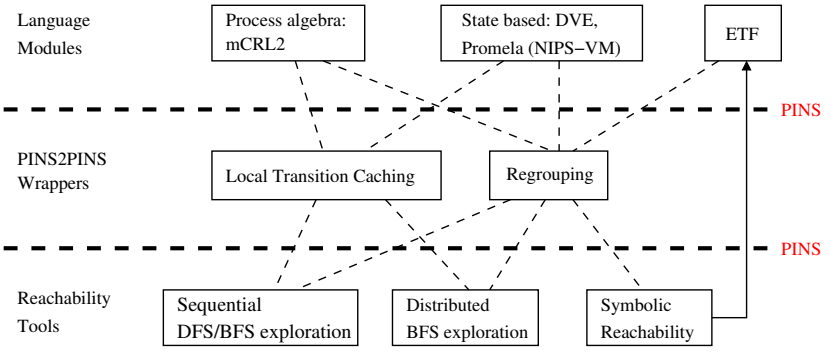


Fig. 1. Architectural Overview of PINS-Based Tools

from *both*, input language and the actual model checking paradigm. Thus, this opens new opportunities for research of reusable and composable implementations of model checking algorithms and optimizations.

2 Architecture: A Partitioned Next State Interface (PINS)

In order to separate specification languages from model checking algorithms, many enumerative, on-the-fly model checkers are based on some *next-state* interface. It provides transitions between otherwise opaque and monolithic states. For example, the OPEN/CÆSAR interface [1] has been underlying the success of the CADP toolkit [2].

The unifying concept in LTSMIN is an improvement of this interface, which we call PINS, an **I**nterface based on a ***Partitioned*** **N**ext-**S**tate function. PINS connects language modules to analysis algorithms. The language modules compute for each specification a static dependency matrix, and implement a next-state function reflecting the operational semantics. The analysis algorithms access this abstraction of the specification, which still captures sufficient combinatorial structure to enable huge state space reductions. The key feature to this is the possibility to obtain transitions between subvectors. Due to lack of space, full details are provided elsewhere [3,4].

In a nutshell, a state for PINS is a vector of N slots, where a single slot can represent anything. The transition relation is split disjunctively into K groups. The $K \times N$ Boolean *dependency matrix* then denotes on which slots each group might depend. Dynamically, a dependency matrix is exploited as follows. Assume that transition group k depends on a short vector of state slots $\langle x_1, \ldots, x_\ell \rangle$ only. PINS next state function operates on this short vector, yielding a short next state, say $\langle y_1, \ldots, y_\ell \rangle$. Note that this result can be reused for many concrete states. By this single call we found a set of transitions on long state vectors: $\langle x_1, \ldots, x_\ell, a_{\ell+1}, \ldots, a_N \rangle \rightarrow \langle y_1, \ldots, y_\ell, a_{\ell+1}, \ldots, a_N \rangle$.

Finally, some optimizations can be expressed purely as transformations of the PINS matrix, also rewiring next-state calls. Such building blocks are implemented once, but all combinations of specification languages and analysis tools can benefit (Fig. 1).

The LTSMIN toolset consists of 28,000 lines of C Code.[2] The interfacing code for the supported frontends (DVE, NIPS, μCRL, mCRL2, our own ETF, Sec. 3.3) consists of only 200–500 lines each. The majority of code is in the three reachability tools, their support data structures, PINS2PINS wrappers, and the TORX [5] and CADP [1] connectors. Taken together, this yields 25 tool combinations, in addition to the minimization tool and various other support tools. The toolset is tested on Linux and MacOS X.

3 Functionality

3.1 Multiple Specification Languages

State-Based Languages. We implemented a language module for the DVE implementation of Barnat el al., giving access to the BEEM benchmark database [6]. Another language module connects the NIPSVM state generator [7], an interpreter for PROMELA, giving access to (pure) SPIN models [8]. The latter module could be refined by making the dependency matrix sparser for global variables and channels, which in general would improve the performance of the reachability tools.

Process Algebras. We have connected the native state generators of the μCRL [9] and mCRL2 [10] toolsets to LTSMIN. Both toolsets specify models in ACP-style process algebra with data, and are heavily used in industrial case studies [9]. They provide expressive ways to model systems, e.g., abstract data types (unbounded numbers, lists, trees), constrained data enumeration, and multi-way handshake communication.

Through the link with LTSMIN, users of all these tools gain for free 100% compatible enumerative, symbolic and distributed model checking tools, as well as compact state space storage formats and minimization tools.

3.2 Reachability and Minimization Tools

We implemented several tools for high-performance state space generation, in particular based on symbolic and distributed model checking. All exploration tools can check safety properties on-the-fly, and produce counter examples upon property violation. Alternatively, full state spaces can be generated and stored for minimization and analysis by external third-party model checkers.

Sequential: Implementations of standard enumerative reachability algorithms, using BFS or DFS search order. These PINS-based tools allow a base-line comparison with the native space generation facilities.

Symbolic: Implementations of symbolic reachability tools. Sets of states are stored as *(binary) decision diagrams*. The state space is computed symbolically by applications of the relational product. More precisely, for any specification language with an enumerative state generator implementing PINS, we automatically obtain a symbolic generator [3,4].

Distributed: Implementations of distributed state space generators, now based on the PINS interface, generalizing our earlier work [11]. This effectively combines the memory of many workstations, also achieving considerable speedups.

[2] Measured with David A. Wheeler's 'SLOCCount'.

PINS2PINS *wrappers:* All generators profit from optimizations in the PINS2PINS layer (Fig. 1). *Local transition caching* is useful for both enumerative generators; *tree compression* [11] is a technique for reducing memory footprint of enumerative generators; and *variable reordering* and *transition regrouping* [3] are useful for the symbolic generator, and in combination with transition caching.

Finally, in case of full state space generation, the LTSMIN toolset includes the distributed minimization tool `ltsmin-mpi` for (strong and branching) bisimulation reduction of labelled transition systems [12]. Also, Orzan's distributed τ-cycle elimination `ce-mpi` [13] tool is included. τ-Cycle freeness in turn admits the use of a simplified distributed minimization algorithm [14] for branching bisimulation. State based equivalences could be easily obtained by modifying the initial partition.

3.3 Tool Interoperability

Besides connecting to native state space generators of various languages (Sec. 3.1), LTSMIN provides converters or interfaces to third party back-end model checkers.

ETF. We defined our own Extended Table Format,[3] which enumerates all short transitions for all groups. It serves as input language of PINS, and as concise output format. E.g., we saw a 0.57 billion state LTS fit in a 1.6 Kb ETF file.

CADP and μCRL. LTSMIN has connections to the well-known CADP toolbox. State spaces can be exported in *binary coded graph* (BCG) format. LTSMIN also implements the CÆSAR/OPEN interface [1] to CADP's on-the-fly model checking and bisimulation algorithms. State spaces can be converted in μCRL's DIR format, allowing to use and compare against *their* implementation of distributed minimization tools.

DIVINE framework. The LTSMIN toolset includes a converter (`etf2dve`) from our ETF format to the input language of the DIVINE toolset [15], DVE. Thus, we obtain access to DIVINE's battery of distributed model checking algorithms. An interesting application is the certification of model checking results, to improve user confidence.

TORX testing tool. LTSMIN implements the TORX RPC interface ($\langle spec \rangle$`2torx`), which allows *test case derivation* with TORX [5] for all PINS language modules. Additionally, JTORX allows checking two specifications for ioco-conformance [16].

GNA tool. In EC-MOAN,[4] the Genetic Network Analyzer [17] exports discrete abstractions of biological ODE models to ETF, and LTSMIN generates their state space for further analysis.

4 Experiments

We performed extensive benchmarking. Precise experimental results go beyond the scope of this tool paper. As illustration, the log-log scatter plot in Fig. 2 shows how distributed and symbolic model checking tools complement each other on selected DVE

[3] `http://fmt.cs.utwente.nl/tools/ltsmin/etf.html`

[4] European FP6 project on biological cell modeling and analysis, see `http://www.ec-moan.org/`

models from the *BEnchmarks for **Explicit** Model Checkers* (BEEM) database [6], ranging from 3×10^6 to 0.57×10^9 states. Each point represents two runs for one specification. The vertical axis indicates the wall-clock time (in seconds) for symbolic reachability (using variable reordering and the chaining heuristic); the horizontal axis denotes the time taken by distributed reachability (on 8×6 cores; with transition caching). The two models near the bottom-right corner are cases where symbolic methods are more than two orders of magnitude faster, whereas for `lift.[78]` and `pgm_protocol.8` the distributed tool is faster by more than factor 10. These are the first reported BDD-based experiments on benchmarks from the BEEM database, whose models are naturally biased towards enumerative methods.

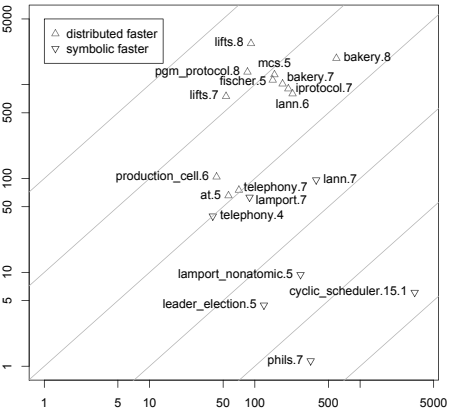


Fig. 2. Wall-clock time in seconds for distributed (x) and symbolic (y) reachability

In INESS,[5] LTSmin is used for the safety analysis of novel railway interlocking specifications. XUML statecharts are translated to MCRL2, and analyzed for safety properties by LTSMIN [18]. Depending on the track layout, we generated state spaces of up to 1.5×10^{11} states directly from MCRL2 models, by means of our symbolic tools.

References

1. Garavel, H.: OPEN/CÆSAR: An open software architecture for verification, simulation, and testing. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 68–84. Springer, Heidelberg (1998)
2. Garavel, H., Mateescu, R., Lang, F., Serwe, W.: CADP 2006: A toolbox for the construction and analysis of distributed processes. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 158–163. Springer, Heidelberg (2007)
3. Blom, S.C.C., van de Pol, J., Weber, M.: Bridging the gap between enumerative and symbolic model checkers. Technical Report TR-CTIT-09-30, University of Twente, Enschede (2009)
4. Blom, S., van de Pol, J.: Symbolic reachability for process algebras with recursive data types. In: Fitzgerald, J.S., Haxthausen, A.E., Yenigun, H. (eds.) ICTAC 2008. LNCS, vol. 5160, pp. 81–95. Springer, Heidelberg (2008)
5. Tretmans, G.J., Brinksma, H.: TorX: Automated model-based testing. In: Hartman, A., Dussa-Ziegler, K. (eds.) First European Conference on Model-Driven Software Engineering, Nuremberg, Germany, pp. 13–43 (2003)
6. 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)
7. Weber, M.: An embeddable virtual machine for state space generation. In: Bošnački, D., Edelkamp, S. (eds.) SPIN 2007. LNCS, vol. 4595, pp. 168–186. Springer, Heidelberg (2007)

[5] European FP7 project on INtegrated European Signalling Systems, `http://www.iness.eu/`

8. Holzmann, G.J.: The model checker Spin. IEEE Trans. Software Eng. 23(5), 279–295 (1997)
9. Blom, S.C.C., Calamé, J.R., Lisser, B., Orzan, S., Pang, J., van de Pol, J., Dashti, M.T., Wijs, A.J.: Distributed analysis with μCRL: a compendium of case studies. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 683–689. Springer, Heidelberg (2007)
10. Groote, J., Keiren, J., Mathijssen, A., Ploeger, B., Stappers, F., Tankink, C., Usenko, Y., Weerdenburg, M.v., Wesselink, W., Willemse, T., Wulp, J.v.d.: mCRL2 toolset. In: Proc. of the IW on Advanced Software Development Tools and Techniques (2008)
11. Blom, S., Lisser, B., van de Pol, J., Weber, M.: A Database Approach to Distributed State-Space Generation. J. Logic Computation (2009) (to appear in print)
12. Blom, S., Orzan, S.: Distributed state space minimization. STTT 7(3), 280–291 (2005)
13. Orzan, S.: On distributed verification and verified distribution. PhD thesis, VU Amsterdam, The Netherlands (2004)
14. Blom, S., van de Pol, J.: Distributed branching bisimulation minimization by inductive signatures. In: Brim, L., van de Pol, J. (eds.) Proc. of 8th Parallel and Distributed Methods in verifiCation. ENTCS, vol. 14, pp. 32–46 (2009)
15. 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)
16. Belinfante, A.: JTorX: A tool for on-line model-driven test derivation and execution. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 266–270. Springer, Heidelberg (2010)
17. de Jong, H., Geiselmann, J., Hernandez, C., Page, M.: Genetic network analyzer: qualitative simulation of genetic regulatory networks. Bioinformatics 19(3), 336–344 (2003)
18. Hansen, H.H., Ketema, J., Luttik, S.P., Mousavi, M.R., van de Pol, J.C.: Towards model checking executable UML specifications in mCRL2. Innovations in Systems and Software Engineering 6(1-2), 83–90 (2010)

`libalf`: The Automata Learning Framework*

Benedikt Bollig[1], Joost-Pieter Katoen[2], Carsten Kern[2], Martin Leucker[3], Daniel Neider[2], and David R. Piegdon[2]

[1] LSV, ENS Cachan, CNRS
[2] RWTH Aachen University
[3] TU München

Abstract. This paper presents `libalf`, a comprehensive, open-source library for learning formal languages. `libalf` covers various well-known learning techniques for finite automata (e.g. Angluin's L^*, Biermann, RPNI etc.) as well as novel learning algorithms (such as for NFA and visibly one-counter automata). `libalf` is flexible and allows facilely interchanging learning algorithms and combining domain-specific features in a plug-and-play fashion. Its modular design and C++ implementation make it a suitable platform for adding and engineering further learning algorithms for new target models (e.g., Büchi automata).

1 Introduction

The common objective of all learning algorithms is to generalize knowledge gained throughout a learning process. In such a process, the learning algorithm is confronted with classified examples. They are utilized to derive some kind of hypothesis which is able to classify new examples in conformance with the examples already seen. Typically, learning algorithms are grouped into *online* and *offline* algorithms. Online learning techniques are capable of actively asking queries to some kind of *teacher* who is able to classify these queries. Offline algorithms, on the other hand, are passively provided with a set of classified examples from which they have to build an apposite hypothesis.

In recent years, learning algorithms have become increasingly popular for various application domains and have been successfully used in different fields of computer science, reaching from robotics over pattern recognition (e.g., in bioinformatics) to natural language recognition. Especially in the area of automatic verification, learning techniques have proved their great usefulness. They were used for minimizing partially specified systems [1], model checking blackbox systems (e.g., [2]), and for improving regular model checking (e.g., [3]). To put it bluntly, automata learning is en vogue.

The need for a unifying framework collecting various types of learning techniques is, thus, beyond all questions. In addition, it is desirable to have possibilities of easily exchanging or extending the implemented learning algorithms to compare assets and drawbacks for certain user applications. For users' convenience a library should provide additional features, such as means for statistical evaluation or loggers. Unfortunately, existing learning frameworks only partly cover these requirements.

The main objective of this paper is to present a new library called the *automata learning framework* (`libalf` for short). `libalf` unifies different kinds of learning

* This work is partially supported by the DAAD (Procope 2009).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 360–364, 2010.

techniques into a single flexible and easy-to-extend library with a clearly structured user interface. We would like libalf to become a comprehensive compendium of learning techniques to which everybody has access and can contribute in a public domain fashion.

2 Related Work

A large number of learning algorithms can be found in the literature. Usually, the most important and influential ones are implemented again and again, but often as *quick-and-dirty* implementations, which are only meant to be a proof-of-concept of the researcher's theoretical work. Typically, this implies a lack of extensibility and comparability as the authors did not have time to bother for a clear, extensible design. We are only aware of two learning libraries that aim for the objectives mentioned above; note that Java PathFinder (cf. [4]) also contains a learning submodule (implementing Angluin's L^* algorithm), but this software seems to be too restricted for most cases.

The LearnLib library [5] allows learning of deterministic finite-state automata. It is available as a dedicated, password-protected server located at the University of Dortmund and can be accessed via the Internet. The LearnLib implements Angluin's L^* algorithm for inferring DFA and some slight variants for deriving Mealy machines.

The *Rich Automata Learning and Testing* library [6] (RALT) has been developed in Java yielding a platform independent solution. It also implements L^* and three relatives for inferring Mealy machines. Regrettably, RALT seems not publicly available.

However, two requirements that seem to be crucial for many user application are clearly missing: Firstly, both libraries are limited to learning Mealy machines in an Angluin setting, but in many environments different learning settings occur. Beyond that, a way to augment the libraries with new learning algorithms, in particular for additional kinds of automata models, is clearly missing. Secondly, as LearnLib can be only accessed remotely and RALT is not available, it seems impossible to assess their performance; in fact, we were not able to experimentally evaluate or benchmark libalf to neither existing library in any appropriate manner. To the best of our knowledge libalf is currently the only available automata learning library that is competitive and flexible enough for real world applications.

3 A Library for Learning Automata: libalf

The libalf library is an actively developed and stable open source library[1] for learning and manipulating formal languages; it puts the emphasis on learning deterministic and non-deterministic finite-state machines, but can be easily augmented with new automata classes (for instance, libalf already supports learning of visibly one-counter automata). As of today, libalf comprises a total of nine learning algorithms, cf. Table 1.

Table 1. Algorithms available in libalf.

Online algorithms	Offline algorithms
Angluin's L^* (2 variants)	Biermann (2 variants)
NL^* [7]	RPNI
Kearns / Vazirani	DeLeTe2
Visibly 1-counter automata [8]	

[1] libalf is freely available on http://libalf.informatik.rwth-aachen.de/

libalf consists of a core C++ library and is complemented by two additional components: liblangen (a library to generate random regular languages) and AMoRE++ (a C++ automata library, among others featuring the antichains algorithm described in [9]). Although written in C++, libalf fits seamlessly into diverse environments: it runs on MS Windows, Linux, and Mac OS (in 32- and 64-bit) and features a platform independent Java interface (using the Java Native Interface JNI). In addition, the so-called *dispatcher* implements a network-based client-server architecture, which allows one to run libalf remotely, e.g., on a high-performance machine.

The key objectives of libalf are *high flexibility* and *simple extensibility*. High flexibility, on the one hand, means that libalf lets the user easily switch between learning algorithms and information sources (often only by changing a single line of code[2]). This allows one to experiment with different learning techniques, making it possible for the user to choose the algorithm best suited for her setting. Moreover, libalf's visualization and logging facilities enable researchers to gain a deeper understanding of the differences of existing and new algorithms.

Simple extensibility, on the other hand, mainly refers to libalf's structured C++ class hierarchy, especially the learning algorithms and automata models. That allows developers to easily enrich libalf with additional features such as new learning algorithms, advanced automata classes, domain-specific optimizations, etc.

Obviously, developing a flexible and easy-to-use library while preserving high extensibility was one of the implementation's most challenging tasks. A comparison of important learning libraries to libalf is given in Table 2.

Table 2. Overview over the most important learning libraries in comparison to libalf

	libalf	LearnLib	RALT
Algorithms	online / offline	online	online
	currently 9	1 (L^*)	1 (L^*)
Hypotheses	DFA, NFA, Mealy, visibly one-counter, etc.	DFA, Mealy	DFA, Mealy
Open source	yes	no	n/a
Availability	C++, Java (JNI)	C++	Java
	source code, binary, dispatcher	via Internet connection only	n / a
Specifics	filters, normalizers, statistics, visualization	filters, statistics, visualization	visualization

Technical Details. In libalf words $w \in \Sigma^*$ (i.e., *queries*) are represented as lists of symbols, where each symbol is a 32-bit integer. Thus, the maximal size $|\Sigma|$ of an alphabet Σ is 2^{32}. For hypotheses, on the other hand, libalf provides generic but simple interfaces such that new automata classes can easily be added. However, the AMoRE++ library can be used if a more powerful automata library is needed.

libalf's main components are the *learning algorithms* and the so-called *knowledgebase*. The knowledgebase is an efficient storage for language information and collects *queries* and *classifications* thereof; in libalf a classification can be any C++ object, but in most algorithms it is a Boolean value. Using an external storage has the advantage of being independent of the choice of the learning algorithm. So it becomes possible to quickly interchange different learning algorithms or run them (even concurrently) on the basis of the same knowledgebase (i.e. queries are only conducted once

[2] Visit our website for a Java online demo on how to employ libalf in a user application.

and are then available to any learning algorithm). Clearly, this helps the user experiment and decide which algorithm to use in her specific setting.

Additionally, libalf features two types of domain-specific optimizations: *filters* and *normalizers*. Filters are a means for reducing the number of queries asked to the teacher. The idea is that in many cases the classification of a query can be decided without consulting the teacher just by applying simple domain-specific knowledge; take, for instance, well-formedness of XML-documents as such a criterion. If a query can already be answered by a filter, it is not passed on to the teacher and the number of queries actually asked to the teacher is reduced. Moreover, filters can be composed by logical connectors (*and*, *or*, *not*).

In contrast, *normalizers* are a means to reduce memory consumption during the learning phase. A normalizer defines a domain-specific equivalence relation $\sim \subseteq \Sigma^* \times \Sigma^*$ over all words and only stores data for one representative of each equivalence class (i.e. data for equivalent queries is only queried and stored once). This does not only reduce the consumed memory, but also the number of queries conducted. By subtyping the respective interface, a user can easily define her own domain-specific optimizations.

Finally, libalf comprises auxiliary components to ease application development and debugging: a *logger* (an adjustable logging facility an algorithm can write to), extensive *statistics* and methods to produce GraphViz visualizations. All of libalf's components are designed to be used in a plug-and-play manner and, to this end, no knowledge about the libraries implementation is required.

4 Conclusion

libalf is a new, comprehensive open-source learning framework, which is easy to use and extend. It gathers several on- and offline learning techniques. The main features of our library and other approaches described previously are summarized in Table 2.

Our learning library is currently used and extended for inferring CFMs from MSC specifications [10] and for learning attractor sets in infinite games (D. Neider, RWTH Aachen). Moreover, there are requests for using libalf for searching through source code to find similar code fragments, so-called clones, (E. Jürgen, TU Munich) and for learning black box systems from log files.

For future work, we plan to augment libalf with additional learning algorithms, e.g., learning using homing sequences or Trakhtenbrot's algorithm, and to integrate learning techniques for other important language classes, such as transducers, Büchi automata etc. Another ongoing work puts different learning algorithms in comparison. In this project, we compare different online and offline learning algorithms and evaluate their average time complexity. The results obtained so far look very promising.

References

1. Oliveira, A.L., Silva, J.P.M.: Efficient Algorithms for the Inference of Minimum Size DFAs. Machine Learning 44(1/2), 93–119 (2001)
2. Groce, A., Peled, D., Yannakakis, M.: Adaptive model checking. In: Katoen, J.-P., Stevens, P. (eds.) TACAS 2002. LNCS, vol. 2280, pp. 357–370. Springer, Heidelberg (2002)

3. Habermehl, P., Vojnar, T.: Regular Model Checking Using Inference of Regular Languages. ENTCS 138(3), 21–36 (2005)
4. Giannakopoulou, D., Pasareanu, C.S.: Interface Generation and Compositional Verification in Java Pathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
5. Raffelt, H., Steffen, B., Berg, T., Margaria, T.: LearnLib: a framework for extrapolating behavioral models. STTT 11(5), 393–407 (2009)
6. Shahbaz, M.: Reverse Engineering Enhanced State Models of Black Box Software Components to Support Integration Testing. PhD thesis, Laboratoire Informat. de Grenoble (2008)
7. Bollig, B., Habermehl, P., Kern, C., Leucker, M.: Angluin-Style Learning of NFA. In: IJCAI 2009, pp. 1004–1009. AAAI Press, Menlo Park (2009)
8. Neider, D., Löding, C.: Learning Visibly One-Counter Automata in Polynomial Time. Technical Report AIB-2010-02, RWTH Aachen (January 2010)
9. Wulf, M.D., Doyen, L., Henzinger, T.A., Raskin, J.F.: Antichains: A new algorithm for checking universality of finite automata. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 17–30. Springer, Heidelberg (2006)
10. Bollig, B., Katoen, J.P., Kern, C., Leucker, M.: Learning Communicating Automata from MSCs. IEEE TSE (to appear)

Symbolic Bounded Synthesis*

Rüdiger Ehlers

Reactive Systems Group
Saarland University
ehlers@react.cs.uni-saarland.de

Abstract. Synthesis of finite state systems from full linear time temporal logic (LTL) specifications is gaining more and more attention as several recent achievements have significantly improved its practical applicability. Many works in this area are based on the Safraless synthesis approach. Here, the computation is usually performed either in an explicit way or using symbolic data structures other than binary decision diagrams (BDDs). In this paper, we close this gap and consider Safraless synthesis using BDDs as state space representation. The key to this combination is the application of novel optimisation techniques which decrease the number of state bits in such a representation significantly. We evaluate our approach on several practical benchmarks, including a new load balancing case study. Our experiments show an improvement of several orders of magnitude over previous approaches.

1 Introduction

Ensuring the correctness of a system is a difficult task. Bugs in manually constructed hard- or software are often missed during testing. To remedy this problem, two lines of research have emerged. The first one deals with the verification of systems that have already been built and spans topics such as process calculi and model checking. The second line concerns the automatic derivation of systems that are correct by construction, also called *synthesis*. In both cases, the specification of the system needs to be given, but we can save the work of constructing the actual system in the case of synthesis.

Unfortunately, the complexity of synthesis has been proven to be rather high. For example, when given a specification in form of a property in linear time temporal logic (LTL), the synthesis task has a complexity that is doubly-exponential in the size of the specification [17]. Recently, it has been argued that this is however not a big problem [18] as *realisable* practical specifications typically have implementations that are small, which can be exploited. This observation is used in the context of *bounded synthesis* [18,8], which builds upon the *Safraless synthesis* principle [14]. Here, the LTL specification is converted to a universal co-Büchi

* This work was supported by the German Research Foundation (DFG) within the program "Performance Guarantees for Computer Systems" and the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems" (SFB/TR 14 AVACS).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 365–379, 2010.

word or tree automaton, which is then, together with a bound $b \in \mathbb{N}$, used for building a safety game such that winning strategies in the game correspond to implementations satisfying the specification. The bound in this setting describes the maximum allowed number of visits to rejecting states in the co-Büchi automaton. If there exists an implementation satisfying a given specification, then there exists some bound such that the resulting game is winning.

In practice, the bound required is usually rather small, often much smaller than the number of states in the smallest implementation. This leads to improved running times of implementations following this approach. Consequently, all modern tools for full LTL synthesis publicly available nowadays build upon Safraless synthesis. The first of these, named Lily [11], performs the realisability check in an explicit way. Recently, a symbolic algorithm based on antichains has been presented [8], showing a better performance on larger specifications. Surprisingly, the usage of binary decision diagrams (BDDs), a technique that has skyrocketed the size of the systems that can be handled by model checking tools [15], seems to be unconsidered in this context so far. A possible explanation for this is that the safety games constructed in the bounded synthesis context contain a lot of *counters* with dependencies between them in the transition relation. It has been observed that this can tremendously blow-up the size of BDDs [22,19,3]. Thus, for success using this technique, it is a central requirement that efficient techniques for reducing the number of counters are being used. In this paper we investigate this problem and present such techniques. By taking them together, we can improve upon the performance of previous approaches to full LTL synthesis by several orders of magnitude.

In particular, we show how to split a specification, consisting of assumptions about the environment and guarantees that the system needs to fulfill, into safety and non-safety parts, which can be handled separately in the synthesis game. As for safety properties, no counters are necessary, this reduces the computation time significantly and allows utilising a major strength of BDDs: efficient dealing with automata that run in parallel. Since it has been argued that typical specifications found in practice are mostly of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ for some sets of assumptions A and guarantees G [2,20], both containing LTL formulas, we design our technique to be adapted to this case. As a second contribution, we discuss the efficient encoding of the safety and non-safety parts in BDD-based games. Finally, we show how to adapt the techniques presented to checking the *unrealisability* of a given specification in an efficient way. We evaluate our approach on the benchmarks from [11,8] and also present a new, more complex *load balancing* benchmark that allows for a more meaningful discussion of the practical applicability of our approach.

This paper is structured as follows. In the next section, we briefly discuss the preliminaries and give suitable references for those readers who are not familiar with the fundamentals of bounded synthesis. Then, we show how a specification can be split into safety and non-safety parts without losing soundness or completeness of the synthesis procedure. Section 4 describes how to efficiently encode both parts in a symbolic state space. In Section 5, we continue with the

explanation of how the unrealisability of a specification can also be checked with our approach. Section 6 contains the experimental results of running our prototype tool on the benchmarks from [11] and [8] as well as on a novel load-balancing system case study. We conclude with a summary.

2 Preliminaries

This section describes the fundamentals of the bounded synthesis approach. We choose the notation in a way such that it fits best to the presentation of the new concepts in the remaining sections.

Mealy automata: For the representation of systems to be synthesized, a suitable computation model is required. In this work, we use *Mealy automata* [16]. Formally, a Mealy automaton $\mathcal{M} = (S, I, O, \delta, s_{\text{in}})$ is defined as a 5-tuple with the set of states S, the input set I, the output set O, the transition function $\delta : (S \times I) \to (S \times O)$ and the initial state $s_{\text{in}} \in S$. For the scope of this paper, we assume that the sets S, I and O are finite. We set $I = 2^{\text{AP}_I}$ for some input proposition set AP_I and $O = 2^{\text{AP}_O}$ for some output proposition set AP_O as this facilitates the description of properties of Mealy automata with temporal logic.

Given some input stream $d = d_1 d_2 \ldots \in I^\omega$ to a Mealy automaton, we define the computation of the automaton *induced* by d as $\pi = s_0 s_1 s_2 \ldots \in S^\omega$ s.t. $s_0 = s_{\text{in}}$ and for all *rounds* $j \in \mathbb{N}_0$, we have $\delta(s_j, d_{j+1}) = (s_{j+1}, o)$ for some $o \in O$. Furthermore, the output of $\mathcal{A}$ over d is defined as $\rho = \rho_1 \rho_2 \ldots$ such that for all $j \in \mathbb{N}_0$, we have $\delta(s_j, d_{j+1}) = (s_{j+1}, \rho_{j+1})$. We furthermore say that $w = (d_1 \cup \rho_1)(d_2 \cup \rho_2) \ldots$ is a word induced by $\mathcal{M}$.

Linear time temporal logic (LTL) & universal co-Büchi word automata: For the specification of a system to be synthesized, some description logic is necessary. *Linear time temporal logic (LTL)* has been the predominantly used such logic in previous works. It allows the usage of the *Safraless synthesis approach*, which circumvents the need for constructing deterministic automata from the specification that occurs in other synthesis methodologies.

Due to space restrictions, we do not define LTL and its semantics here but rather refer to [7]. Formulas in LTL can use the temporal operators "G" (globally), "F" (finally), "X" (next time) and "U" (until). We say that some automaton $\mathcal{M}$ satisfies an LTL formula ψ if for all words $w = w_1 w_2 \ldots$ induced by $\mathcal{M}$, we have $w \models \psi$. Some LTL formulas are also called *safety properties*; this is the case if for every word w not satisfying the property, there exists some prefix w' of w such that no word having the same prefix satisfies the property.

Formulas in LTL can be transformed into equivalent *universal co-Büchi word automata* (UCW), i.e., given an LTL formula ψ, a UCW $\mathcal{A}$ of size at most exponential in $|\psi|$ can be obtained such that for every Mealy automaton $\mathcal{M}$, all runs induced by $\mathcal{M}$ are accepted by $\mathcal{A}$ if and only if all words induced by $\mathcal{M}$ satisfy ψ.

We define universal co-Büchi word automata as five-tuples $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{in}}, F)$ with a set of states Q, an alphabet Σ, a transition function $\delta : Q \times \Sigma \to 2^Q$,

an initial state $q_{\text{in}} \in Q$ and some set of rejecting states $F \subseteq Q$. Given a word $w = w_1 w_2 \ldots \in \Sigma^\omega$, we say that a sequence $\pi = \pi_0 \pi_1 \pi_2 \ldots \in Q^\omega$ is a run of $\mathcal{A}$ over w if $\pi_0 = q_{\text{in}}$ and for all $j \in \mathbb{N}_0$, $\pi_{j+1} \in \delta(\pi_j, w_{j+1})$. A word w is *accepted* by $\mathcal{A}$ if for all runs π of $\mathcal{A}$ over w, we have $\inf(\pi) \cap F = \emptyset$ for inf denoting the function that maps a sequence onto the set of elements that occurs infinitely often in it. We say that a Mealy automaton $\mathcal{M}$ is accepted by $\mathcal{A}$ if all words induced by $\mathcal{M}$ are accepted by $\mathcal{A}$. Due to the finiteness of Mealy automata, if $\mathcal{M}$ is accepted by $\mathcal{A}$, there exists a finite upper bound $b(\mathcal{M}, \mathcal{A})$ on the number of rejecting states visited on the runs of $\mathcal{A}$ on any word induced by $\mathcal{M}$. This bound is always at most $|F| \cdot |S|$ for S being the state set of the Mealy automaton [18].

Safety games: Given some universal co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_{\text{in}}, F)$ with $\Sigma = I \times O$ and some bound $b \in \mathbb{N}$, we can build a two-player *safety game* $\mathcal{G}$ such that player 1 wins the game if and only if there exists some Mealy automaton $\mathcal{M}$ over the inputs I and outputs O with $b(\mathcal{M}, \mathcal{A}) \leq b$ [18].

Formally, we define safety games as tuples $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{\text{in}}, v_F)$ with some vertex set (also called *state space* in the context of synthesis) V, some action set Σ_0 for player 0, some action set Σ_1 for player 1, some total edge function $\delta : V \times \Sigma_0 \times \Sigma_1 \to V$, some initial vertex v_{in} and some final vertex v_F. We require that v_F is *absorbing*, i.e., for all $x \in \Sigma_0 \times \Sigma_1$, $\delta(v_F, x) = v_F$. A *decision sequence* is an infinite sequence $\rho = \rho_0 \rho'_0 \rho_1 \rho'_1 \ldots$ such that for all $j \in \mathbb{N}_0$, $\rho_j \in \Sigma_0$ and $\rho'_j \in \Sigma_1$. Such a decision sequence induces an infinite *play* $\pi = \pi_0 \pi_1 \ldots$ in $\mathcal{G}$ such that $\pi_0 = v_{in}$ and for all $j \in \mathbb{N}_0$, we have $\delta(\pi_j, \rho_j, \rho'_j) = \pi_{j+1}$. We call plays winning for player 1 (the *system player*) if there does not exist some $j \in \mathbb{N}$ such that $\pi_j = v_F$. For the scope of this paper, we also need *reachability games*; in these, player 1 wins a play if there exists some $j \in \mathbb{N}$ such that $\pi_j = v_F$.

Safety games are *memoryless determined*, i.e., if and only if player 1 wins the game, there exists some function $f : V \times \Sigma_0 \to \Sigma_1$ such that for all decision sequences $\rho = \rho_0 \rho'_0 \rho_1 \rho'_1 \ldots$ with corresponding plays $\pi = \pi_0 \pi_1 \ldots$, if $\rho'_j = f(\pi_j, \rho_j)$ for all $j \in \mathbb{N}_0$, then π is winning for player 1. The situation for player 0 is dual.

Given some bound $b \in \mathbb{N}$, some input and output alphabets Σ_0/Σ_1 and some universal co-Büchi word automaton $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ with $\Sigma = \Sigma_0 \times \Sigma_1$, the corresponding (classical) *synthesis game* is defined as $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{\text{in}}, v_F)$ with a vertex set V comprising all functions mapping the states in Q onto $\{\bot, 0, 1, \ldots, b\}$. The vertices of the game encode in which states of $\mathcal{A}$ a run of the automaton corresponding to the input/output played by the players so far could be. All such states have a numeral value assigned, whereas the others are mapped to $\bot$. The numeral value represents how many rejecting states have been visited at most along such a run so far (the so-called *counters*). For details of this approach, the reader is referred to [18].

We have defined safety games in a way such that we can efficiently extract a Mealy automaton $\mathcal{M}$ satisfying $\mathcal{A}$ from a winning strategy f. We define the *winning region* of $\mathcal{G}$ to be the largest subset of vertices in V such that for setting v_{in} to any of these, the game is winning for player 1.

Binary decision diagrams: For representing sets of vertices and the transition relation in safety games symbolically, we use *reduced ordered binary decision*

diagrams (BDDs) [4,5], which represent characteristic functions $f : 2^V \to \mathbb{B}$ for some finite set of variables V. Since they are well-established in the context of formal verification, we do not describe their details here but rather treat them on an abstract level and state the operations on them that we use. For a comprehensive overview, see [5]. Given two BDDs f and f', we define their conjunction and disjunction as $(f \wedge f')(x) = f(x) \wedge f'(x)$ and $(f \vee f')(x) = f(x) \vee f'(x)$ for all $x \subseteq V$. The negation of a BDD is defined similarly. Given some set of variables $V' \subseteq V$ and a BDD f, we define $\exists V'.f$ as a function that maps all $x \subseteq V$ to **true** for which there exists some $x' \subseteq V'$ such that $f(x' \cup (x \setminus V')) =$ **true**. Dually, we define $\forall V'.f \equiv \neg(\exists V'.\neg f)$. Given two ordered lists of variables $L = l_1, \ldots, l_n$ and $L' = l'_1, \ldots l'_n$ of the same length, we furthermore denote by $f[L/L']$ the BDD for which some $x \subseteq V$ is mapped to true if and only if $f(x \setminus \{l'_1, \ldots, l'_n\} \cup \{l_i \mid \exists 1 \leq i \leq n : l'_i \in x\}) =$ **true**.

2.1 Differences to Other Works

In contrast to previous works on Safraless synthesis, we give a simplified presentation here, which relies on universal co-Büchi word automata (UCW) instead of co-Büchi tree automata [14,18] or transition-based UCWs [8].

Furthermore, the definition of safety games differs from the one used when synthesizing Moore automata. First of all, we assume that player 0 (the environment) does the first move instead of player 1 (the system player). This way, the game model corresponds to the behaviour of Mealy automata. This slightly changes the semantics of the LTL formulas for synthesis. For example, the specification $G(r \leftrightarrow g)$ for the input atomic proposition (AP) set $\{r\}$ and the output AP set $\{g\}$ is realisable, whereas for the reversed order of input and output used in previous works, it is unrealisable. The intuition of this change is that this reduces the number of next-time LTL operators necessary for practical specifications, thus reducing the size of the UCW for the specification and the synthesis time needed in total. Nevertheless, the techniques presented in this paper are equally applicable to Moore automata synthesis.

Additionally, the fact that we do not have vertex sets for both players 0 and 1 in the game allows us to simplify the game solving process and also saves bits for the state sets in a symbolic game solving process. Given a safety game $\mathcal{G} = (V, \Sigma_0, \Sigma_1, \delta, v_{in}, v_F)$, we build a BDD B^δ corresponding to δ over the four lists of variables $\{pre, in, out, post\}$ such that for all $q, q' \in Q$, $i \in \Sigma_1$ and $o \in \Sigma_2$, by abuse of notation, $B^\delta(q, i, o, q') =$ **true** if and only if $\delta(q, i, o) = q'$ (for some encoding of the states, inputs and outputs into the BDD variables). Using B^δ and some BDD B^F over the variables in *post* mapping only v_F to **true**, we can compute the winning region of $\mathcal{G}$ as $\nu X.X \wedge (\forall in.\exists out, post.(B^\delta \wedge X[post/pre] \wedge (\neg B^F)))$ for ν denoting the greatest fixed point operator.

3 Safety and Non-safety: Splitting the Specification

In this section, we explain how to decompose an LTL specification being subject to synthesis in a way such that non-safety and safety properties can be treated in

parallel. Recall that we assume that the specification is written in the form $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$. In the classical bounded synthesis approach, ψ is transformed to a UCW which in turn is converted to its induced safety game for some given bound. Here, we propose a slightly different approach. Instead of building one single game from the specification, we split the latter into parts, build individual games for each of the parts and then take their parallel composition to obtain a *composite game*. This has several advantages:

1. It has been observed [8] that the time to compute a UCW from an LTL formula is a significant part of the overall realisability checking time. By splitting the specification beforehand, building a monolithic UCW is avoided, resulting in a lower total computation time.
2. Taking the parallel composition of multiple game structures can be done in a relatively efficient way when using BDDs for solving the composite game.
3. The state spaces of games corresponding to safety properties do not need the counters that are employed in the bounded synthesis approach. Thus, by decomposing the specification into safety and non-safety parts, we can save counters, which in turn reduces the computation time further.

In order to obtain a valid decomposition scheme, the resulting game must be winning for player 1 (the system player) in the same cases as before, i.e., if and only if either a safety or non-safety assumption is violated or all guarantees are fulfilled. The technique presented in the following does not preserve the smallest bound b such that the specification is fulfillable (as the bound depends on the syntactic structure of the UCW). However, the method proposed is still sound and complete, i.e., if and only if there exists a bound b such that the safety game induced by the UCW for the overall specification and b is winning for player 1, there exists some bound for the non-safety part of the specification and the technique presented in this section such that the resulting game is winning for player 1.

In [20], the authors propose a method to solve a generalised parity game for a specification of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ as stated above successively. They first build games for the safety assumptions and guarantees, strip the non-winning parts (for the system player) from them and compose them with games for the remaining parts of the specification. For completeness of this methodology, the non-safety assumptions however must not have any effect on the fulfillability of the safety guarantees. In general, we cannot assume this; we thus propose a different method here that is based on introducing some kind of *signal* into the game that links the safety guarantees and the non-safety part of the specification.

We start by splitting the specification $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ into four sets of LTL formulas: the safety assumptions A_s, the safety guarantees G_s, the non-safety assumptions A_n, and the non-safety guarantees G_n. Then, we build a reachability game $\mathcal{G}_1$ for the safety assumptions that is won by player 1 if some assumption in A_s is violated. For the next step, we add one bit to the output atomic proposition set of the system to be synthesized; let its name be safe_g. We build a safety game $\mathcal{G}_2$ from the safety guarantees G_s that is won by player

1 if safe_g always represents whether one of the safety guarantees has already been violated. For the non-safety part, we take the *modified specification* $\psi' = (\bigwedge_{a \in A_n} a) \rightarrow (\bigwedge_{g \in G_n} g \wedge G(\text{safe}_g))$ and convert it to a UCW $\mathcal{A}$. Given a bound $b \in \mathbb{N}$ and having prepared $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{A}$, we can now build the composite game $\mathcal{G}$:

1. We take $\mathcal{A}$ and b and build the corresponding bounded synthesis safety game. Let its name be $\mathcal{G}_3$.
2. We define the overall synthesis game $\mathcal{G}$ as the parallel composition of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$, i.e., the vertex set is the product of the individual vertex sets and the transition relation is defined such that the games $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ are played in parallel (over the same inputs and outputs). We say that $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ are *components* of $\mathcal{G}$.
3. Let q_F^1, q_F^2 and q_F^3 be the final vertices of the games $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$, respectively. We define a play π in $\mathcal{G}$ to be winning for the system player if either q_F^1 is visited at some point on π or q_F^2 and q_F^3 are never visited.

We obtain the following result:

Theorem 1. *For every LTL specification* $\psi = \bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$*, there exists some bound* $b \in \mathbb{N}$ *such that the composite game* $\mathcal{G}$ *built from* ψ *and* b *as defined above is won by the system player* 1 *if and only if there exist some bound* $b' \in \mathbb{N}$ *such that the (classical) safety synthesis game induced by the UCW corresponding to* ψ *and* b' *is winning for player* 1*.*

Proof. By examining the possible causes for winning/losing the synthesis games, the correctness of the claim can easily be seen. □

Let B_1^F be a BDD over the set of variables *pre* representing the final vertices of the game $\mathcal{G}_1$ and B_2^F and B_3^F be BDDs over *post* for the final vertices of $\mathcal{G}_2$ and $\mathcal{G}_3$, respectively. For B^δ being the BDD representing the transition relation of $\mathcal{G}$, we can obtain the winning region of player 1 by computing (for μ denoting the least fixed point operator):

$$V = \mu Y.Y \vee B_1^F \vee (\forall in.\exists out, post.B^\delta \wedge Y[post/pre])$$
$$W = \nu X.V \vee (X \wedge (\forall in.\exists out, post.B^\delta \wedge X[post/pre] \wedge (\neg B_2^F) \wedge (\neg B_3^F)))$$

In these equations, V represents the states that are winning due to the fact that the system player can choose a sequence of decisions such that some safety assumption is not fulfilled; W is the winning region for player 1 in $\mathcal{G}$.

We can simplify the computation by taking $V = B_1^F$, making the composite game essentially a safety game. To see this, consider a state in the game in which some guarantee has just been violated but that is still winning as from that state onwards, the system player can force the other player into a state in which also some safety assumption is violated. As the game is finite, there is an upper bound of k steps for some $k \in \mathbb{N}$ on the length of such a bridging path in the game. By increasing the bound used for building $\mathcal{G}_3$ by k, it can be made sure that q_1^F is reached before q_3^F is visited, making the game also winning with

the modified definition for V. We use this simplification for our implementation to be described in Section 6 as it facilitates the extraction of winning strategies from the game.

4 Encoding Bounded Synthesis in BDDs

The efficiency of solving games using BDDs heavily depends on a smart encoding of the state space into the BDD bits. As already stated, for a symbolic solution of a safety game, four groups of BDD variables are needed: two groups for the game vertices (*pre* and *post*), one for the input to the system and one for the output. As we defined the input as $I = 2^{\mathrm{AP}_I}$ and the output as $O = 2^{\mathrm{AP}_O}$ for the scope of this paper, a straight-forward boolean encoding of I and O for usage in the BDDs exists: we allocate one BDD bit for each element of AP_I and AP_O. It remains to find a suitable encoding for the state space of the game.

First of all, if the state space is the product of some smaller state spaces, we can parallelise the problem; for example, if $V = V_1 \times V_2 \times \ldots \times V_m$ for some $m \in \mathbb{N}$, we can find good encodings for each of the state spaces $V_1, \ldots, V_m$ individually. We are thus able to handle the state space encodings of $\mathcal{G}_1$, $\mathcal{G}_2$ and $\mathcal{G}_3$ (as defined in the previous section) separately.

4.1 The Non-safety Part

Recall that in the context of bounded synthesis, the safety game induced by a UCW for a given bound b has a certain property: the state space consists of all functions mapping the states of the UCW onto $\{\bot, 0, 1, \ldots, b\}$ for b being the chosen bound. For each state, we can encode the value the function maps to individually. For the scope of this paper, we define the following encoding for this counter set $\{\bot, 0, 1, \ldots, b\}$: we use $\lceil \log_2(b+1) \rceil + 1$ bits. One bit is used for representing whether the value equals $\bot$, the remaining bits represent the standard binary encoding of the numeral (if given). Taking an extra bit for the $\bot$ value has the advantage of obtaining smaller BDDs in most cases as this value appears very often in the definition of the transition relation.

We also use and propose two additional tricks. First of all, the games defined in the previous section are built in a way such that they permit one type of non-determinism: we can allow the system player to choose a successor state from a set of possible ones. If the system player can do this in a greedy way, i.e., the non-determinism can be resolved after each input/output cycle without losing completeness, the game semantics remain unchanged. For bounded synthesis, we can thus relax the transition relation slightly: we allow the system player to increase her counters in addition to the counter increases imposed by visits to rejecting states. We also allow her to set some counters from $\bot$ to some arbitrary other value. This *non-minimality* [1] of the transition relation typically decreases the size of its symbolic encoding. A similar idea was also pursued by Henzinger et al. [10] for simplifying the process of automaton determinisation.

As a second trick, we can use some automata-theoretic argument for not having to store counters for certain states. Let a *strongly connected component*

(SCC) in a UCW be a maximal set of states such that there exist sequences of transitions between all pairs of states in the SCC. It is well-known that every infinite run of a UCW $\mathcal{A}$ enters a strongly connected component in $\mathcal{A}$ after a finite number of steps that it never leaves again. It is accepting if and only if in this last SCC, rejecting states are visited only finitely often. This fact gives rise to an optimisation idea: for transient states or states in SCCs without rejecting states, we do not really need counters: we can assume that the counter corresponding to such a state is always reset to 0. We call those states in $\mathcal{A}$ *transient* that can only be visited once on every run of the automaton. Thus, only one bit is needed for such states instead of $\lceil \log_2(b+1) \rceil + 1$ bits. This modification does not alter the soundness or completeness of the overall synthesis procedure. Additionally, as some counters are now reset on some transitions, in practice we often have the situation that for realisable specifications, the number of counter bits per remaining state necessary for finding out that the specification is realisable is also less.

4.2 The Safety Part

For the encoding of the game components corresponding to safety assumptions and guarantees, we state two different, straight-forward methods, which we explain in the following. The first method only works for *locally checkable properties* and is usually more efficient than the second one in this case, whereas the latter method is capable of handling arbitrary safety properties.

Smart encoding of locally checkable properties: If an LTL property is of the form $\psi = G(\phi)$ with a formula ϕ in which the only temporal operator occurring is X, then ψ is a *locally checkable property* [12]. Let k be the deepest nesting of the X operator in ϕ. For checking the satisfaction of such a property along a trace, it suffices to store whether the property has already been violated, the last k input/outputs (also called *history*) and the current round number (with the domain $\{0, 1, \ldots, k-1, \geq k\}$). Then, in every round with a number $\geq k$, we update whether the specification is already falsified with the input and output in the last k rounds and the current round. For encoding the round number in a symbolic way, we use a binary representation.

Encoding such a property in this way has some advantages: First of all, the encoding proposed is canonical. Furthermore, multiple properties can share the information stored in the game state space this way, so we can recycle the stored information for all such locally checkable safety properties. Note that it is possible to reduce the number of bits necessary for storage by leaving out the history bits not needed for checking the given properties.

The general method: Safety properties have equivalent *syntactically safe* UCW, i.e., in the UCW, all rejecting states are absorbing. In this case, the UCW can be determinised by the power set construction. Thus, we can assign to each state in the universal automaton a state bit which is set to 1 whenever there is a run from the initial state to the respective state encoded by the bit for the input/output played by the players during the game so far.

This method is applicable to all safety properties but requires the computation of a universal co-Büchi automaton having the property stated above. While it has been observed that checking if a property is safety is not harder than building an equivalent universal co-Büchi automaton [13], it is not guaranteed that typical procedures for constructing UCWs from LTL properties yield automata that have this property. For conciseness, we use a simplified approach in our actual implementation. If the procedure employed for converting an LTL formula into a UCW yields a UCW for which all rejecting states are absorbing or transient, we declare the property as being safety and otherwise treat it as a non-safety property. While we may miss safety properties this way, the soundness of the overall approach is preserved.

5 Checking Unrealisability

So far, we have only dealt with the case that we want to prove realisability of a specification. If a specification is unrealisable, then for no bound $b \in \mathbb{N}$, the safety game induced by the bound and the specification is won for the system player. Thus, an implementation of our approach, which would typically increase the bound successively until the induced safety game is winning for the system player, does not terminate in this case. In [8], it is described how the bounded synthesis approach can be used for detecting unrealisability quickly anyway: we simply run the synthesis procedure both on the original specification as well as on the negated specification with swapped input and output in parallel. One of these runs is guaranteed to terminate. Whenever this happens, we can abort the other run. This results in an decision procedure for the overall problem.

When applying the optimisations from this paper, this idea is not directly usable, as when negating the specification, the result is not again of the form $\bigwedge_{a \in A} a \rightarrow \bigwedge_{g \in G} g$ for some sets of assumptions A and guarantees G. Instead, checking if the environment player wins can be done by swapping input and output, negating only the modified specification, and making the final states of $\mathcal{G}_1$ losing for player 1 instead of winning. Then, player 1 (which is now the environment player) wins only if the safety assumptions are fulfilled, the safe_g bit always represents if a safety guarantee has already been violated, and the negated modified specification is fulfilled (with respect to the given bound).

Using the notations from Section 3, after replacing $\mathcal{G}_3$ with a game corresponding to the negated modified specification, we can compute the set of winning states for the environment player by:

$$W = \nu X.X \wedge (\neg B_1^F) \wedge (\exists in.\forall out.\exists post.B^\delta \wedge X[post/pre] \wedge (\neg B_2^F) \wedge (\neg B_3^F))$$

6 Experimental Results

We implemented our symbolic bounded synthesis approach in C++ with the BDD library CUDD v.2.4.2 [21], using dynamic variable reordering. The prototype tool assumes that the individual guarantees and assumptions are given

separately. The first step in the computation is to split non-safety properties from safety ones. For this, the tool calls the LTL-to-Büchi converter LTL2BA v.1.1 [9] on the negations of the properties to obtain equivalent universal co-Büchi word automata. As described in Section 4.2, we then check if the automata obtained are syntactically safe. Locally checkable properties are converted to games using the procedure specialised in this case, all other safety properties are treated by the general procedure given. The UCW corresponding to the modified non-safety part of the specification (as described in Section 3) is again computed by calling LTL2BA on it. The last step for realisability checking is to solve the composite games built for a successively increasing number of counter bits per state in the UCW until the game is winning for the system player. We always start with two bits.

We always check for realisability and unrealisabilty of the given specification simultaneously, as described in the previous section. In case of realisability, we extract an implementation that fulfills the specification. We do this in a fully symbolic way: the first step is to compute the winning region of the game and identify state bits that have a fixed value throughout all winning plays. These state bits are removed. Then, we restrict the transition relation to moves by the system player for which the lexicographically minimal next winning state is chosen (for some order of the state bits). We do the same for the output bits, i.e., for some order of the output bits, we restrict the resulting transition relation to lexicographically minimal output bit valuations (with respect to the remaining choices for the system player). As a result, the transition relation is weakened in a way such that there is precisely one combination of next state and output bit valuation left for every reachable state and input variable assignment, making the behaviour of player 1 deterministic. The remaining game graph is, together with the specification, converted to a NuSMV [6] model. This allows running NuSMV to verify the correctness of the models produced.

All computation times given in the following are obtained on a Sun XFire computer with 2.6 Ghz AMD Opteron processors running an x64-version of Linux. All tools considered are single-threaded. We restricted the memory usage to 2 GB and set a timeout of 3600 seconds. The running times for our tool always include the computation times of LTL2BA.

6.1 Performance Comparison on the Examples from [11,8]

We compare our prototype implementation with the only other currently publicly available tools for full LTL synthesis, namely Lily v.1.0.2 [11] and Acacia v.0.3.9 [8]. In the following, for Acacia as well as our prototype tool, we only give running times for the non-realisability check if the property is not realisable and the realisability check and model synthesis if the property is realisable.

The 23 mutex variations used as examples in [11,8] are a natural starting point for our investigation. For usage with our tool, we adapted these examples to the Mealy-type computation model used in this work (as described in Section 2.1) by prefixing all references to input variables with a next-time operator. For these 23 examples, Lily needed 54.35 seconds of computation time (of which 44.25 seconds

Table 1. Comparison of the running times of Acacia and our prototype tool (in seconds) on the scalable example no. 3 from [8]. In this table, we denote timeouts by "t/o" and running out of memory by "m/o". As Lily performs worse than Acacia on this benchmark, we did not include Lily in this comparison.

# of Clients:	1	2	3	4	5	6	7	10	14	15	20	21	22
Acacia running times:	0.9	2.0	4.0	9.8	47.3	506.5	m/o	m/o	m/o	m/o	m/o	m/o	m/o
Prototype running times:	0.3	0.7	0.6	1.9	0.9	4.6	3.0	651.5	491.0	t/o	1909.0	t/o	t/o

were devoted to computing the automata from the given specifications). Acacia in turn finished the task in 53.71 seconds (including 42.2 seconds for building the automata). Our prototype implementation had a total running time of about 19.41 seconds. As computing the automata from the specification parts is not a pure preprocessing step in our prototype, we do not split up the total running time here.

In [8], the authors also modify one of these examples in order to be scalable. Table 1 contains the respective results for this example.

6.2 A Load Balancing System

For evaluating the techniques presented in this paper in a more practical context, we present an example concerning a *load balancing unit* distributing requests to a fixed number of servers. Such a unit typically occurs as a component of a bigger system which in turn utilises it for scheduling internal requests. We demonstrate how a synthesis procedure can be used in the early development process of the bigger system in order to systematically engineer the requirements of the load balancer. Using a synthesis tool in this context makes it possible detect errors in the specification that result in unrealisability as early as possible. We start by stating the fundamental properties of the load balancing system and finally tune it towards serving requests to the first server in a prioritised way. After each added specification/assumption, we run our example implementation in order to check if the specification is still realisable.

The following list contains the parts of the specification. Table 2 gives the running times of our tool and Acacia for the respective sets of assumptions and guarantees and some numbers of clients $n \in \{2, \ldots, 9\}$. The system to be synthesized uses the input bits $r_0, \ldots, r_{n-1}$ for receiving the information whether some server is sufficiently under-utilised to accommodate another task and the output bits $g_0, \ldots, g_{n-1}$ for the task assignments. An additional input *job* reports on an incoming job to be assigned. For usage with Acacia, all occurrences of output variables in the specification have been prefixed with a next-time operator to take into account the different underlying computation model.

1. *Guarantee:* Non-ready servers are never bothered: $\bigwedge_{0 \leq i < n} G(g_i \rightarrow r_i)$
2. *Guarantee:* A task is only assigned to one server: $\bigwedge_{0 \leq i < n} G(g_i \rightarrow (\bigwedge_{j \in \{1,\ldots,n\} \setminus \{i\}} \neg g_j))$
3. *Guarantee:* Every server is used infinitely often: $\bigwedge_{0 \leq i < n} GF(g_i)$

Table 2. Running times of Acacia ("A") and our prototype tool ("P") for the subproblems defined in Section 6.2 for $n \in \{2, \ldots, 9\}$. For each combination of assumptions and guarantees, it is reported whether the specification was satisfiable (+/-), how many counter bits per state in the UCW were involved at the end of the computation (only for our prototype tool) and how long the computation took (in seconds). We left out the Lily tool as it is not competitive on the load balancing example.

Tool	Specification / # Clients	2	3	4	5	6	7	8	9
P	1	+ 2 0.6	+ 2 0.6	+ 2 0.2	+ 2 1.3	+ 2 0.2	+ 2 0.3	+ 2 0.2	+ 2 0.3
A		+ 0.3	+ 0.4	+ 0.6	+ 0.9	+ 1.5	+ 2.7	+ 5.3	+ 12.1
P	$1 \wedge 2$	+ 2 0.4	+ 2 0.3	+ 2 0.6	+ 2 0.6	+ 2 0.7	+ 2 0.6	+ 2 0.6	+ 2 0.7
A		+ 0.3	+ 0.3	+ 0.4	+ 0.4	+ 0.6	+ 0.9	+ 1.6	+ 3.1
P	$1 \wedge 2 \wedge 3$	- 2 0.5	- 2 0.5	- 2 0.5	- 2 0.5	- 2 0.7	- 2 1.0	- 2 6.9	- 2 73.9
A		- 19.2	- 475.6	timeout	timeout	timeout	timeout	timeout	timeout
P	$1 \wedge 2 \wedge 4$	+ 2 0.3	+ 3 0.4	+ 3 0.9	+ 4 65.5	+ 4 104.6	+ 4 990.3	timeout	timeout
A		+ 0.6	+ 1.3	+ 8.7	+ 277.9	timeout	timeout	timeout	timeout
P	$1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.2	- 2 0.7	timeout	timeout	timeout	timeout	timeout	timeout
A		- 163.4	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.2	- 2 0.7	- 2 3244.1	timeout	timeout	timeout	timeout	timeout
A		- 175.3	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 4 \wedge 5$	- 2 0.5	- 2 1.1	timeout	timeout	timeout	timeout	timeout	timeout
A		- 190.7	timeout	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8$	+ 2 0.3	+ 3 0.6	+ 3 2.4	+ 4 20.7	+ 4 368.6	timeout	timeout	timeout
A		+ 7.5	+ 69.0	+ 357.4	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	- 2 0.3	- 2 0.2	- 2 0.3	- 2 1.0	- 2 16.8	- 2 449.1	timeout	timeout
A		- 48.8	- 2133.5	timeout	timeout	timeout	timeout	timeout	timeout
P	$6 \wedge 7 \wedge 10 \rightarrow 1 \wedge 2 \wedge 5 \wedge 8 \wedge 9$	+ 2 0.4	+ 2 0.8	+ 3 118.7	timeout	timeout	timeout	timeout	timeout
A		+ 26.9	+ 295.8	timeout	timeout	timeout	timeout	timeout	timeout

Note that the guarantees 1,2 and 3 cannot be fulfilled at the same time as some server might not report when it is ready. Therefore, we replace the third part of the specification and continue:

4. *Guarantee:* Liveness of the system: $\bigwedge_{0 \leq i < n} GF(r_i) \rightarrow GF(g_i)$
5. *Guarantee:* Only jobs that actually exist are assigned: $G((\bigvee_{0 \leq i < n} g_i) \rightarrow job)$.

Again, the guarantees 1, 2, 4 and 5 are unrealisable in conjunction as the *job* signal might never be given. We add the assumption that this is not the case:

6. *Assumption:* There are always incoming jobs: $GF job$

At this point, the system designer gets to know that this added requirement does not fix the unrealisability problem, either. The reason is that the clock cycles in which *job* is set and the cycles in which some server is ready might occur in an interleaved way. We therefore add:

7. *Assumption:* The job signal stays set until the job has been assigned: $G(job \wedge (\bigwedge_{0 \leq i < n} \neg g_i) \rightarrow X(job))$

Note that the specification is still not realisable. The reason is that the ready signal of one server i might always be given after a job assignment to another server j has been given (for some $i \neq j$). If server i then always immediately

withdraws its ready signal, the controller can never schedule a job to server i, contradicting guarantee 4 if both servers i and j are ready infinitely often. We therefore modify guarantee 4 to not consider these cases:

8. *Guarantee:* Every ready signal is either withdrawn or eventually handled: $\bigwedge_{0 \leq i < n} \neg(FG(r_i \wedge \neg g_i))$

We continue by adding a priority to the first server. Note that this breaks realisability again, as server 0 can block the others. As an example, we solve this problem by adding the assumption that server 0 works sufficiently long after it obtains a new job before signalling ready again.

9. *Guarantee*: Server 0 gets a job whenever a job is given and it is ready: $G((\bigvee_{1 \leq i < n} g_i) \rightarrow \neg r_0)$
10. *Assertion*: Server 0 does not report being ready when it gets a task until after an incoming job has been reported on for the next time: $G(g_0 \rightarrow ((\neg job \wedge \neg r_0)\ U\ (job \wedge \neg r_0)))$.

7 Conclusion and Outlook

In this paper, we described the steps necessary to make the bounded synthesis approach work well with symbolic data structures such as BDDs. The key requirement was to reduce the number of counters in the safety games that occur in this approach as much as possible. We performed this task by splitting the specification into safety and non-safety parts and presented an additional trick that allowed stripping some counters from the game component corresponding to the non-safety specification conjuncts. We also discussed efficient encodings of the safety part of the specification into games. Experimental results show a huge speed-up compared to previous works.

One particular issue we did not address in this paper is the extraction of small implementations in the synthesis process for the case that the specification is realisable. Similarly to the observations made in the context of generalised reactivity(1) synthesis, where the expressivity of full LTL is traded against the possibility to use more efficient algorithms for performing the synthesis process, the models produced are often non-optimal [2], i.e., unnecessarily large. Thus, further work will deal with the more effective extraction of winning strategies. While the techniques presented here are already suitable for requirements engineering and prototype extraction, the problem of how to obtain small implementations which can directly be converted to suitable hardware circuits is still open.

References

1. Bloem, R., Cimatti, A., Pill, I., Roveri, M.: Symbolic implementation of alternating automata. International Journal of Foundations of Computer Science 18(4), 727–743 (2007)
2. Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Specify, compile, run: Hardware from PSL. Electr. Notes Theor. Comput. Sci. 190(4), 3–16 (2007)

3. Bozga, M., Maler, O., Pnueli, A., Yovine, S.: Some progress in the symbolic verification of timed automata. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 179–190. Springer, Heidelberg (1997)
4. Bryant, R.E.: Graph-based algorithms for boolean function manipulation. IEEE Trans. Computers 35(8), 677–691 (1986)
5. Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. Inf. Comput. 98(2), 142–170 (1992)
6. Cimatti, A., Clarke, E.M., 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)
7. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. MIT Press, Cambridge (1999)
8. Filiot, E., Jin, N., Raskin, J.F.: An antichain algorithm for LTL realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
9. 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)
10. Henzinger, T.A., Piterman, N.: Solving games without determinization. In: Ésik, Z. (ed.) CSL 2006. LNCS, vol. 4207, pp. 395–410. Springer, Heidelberg (2006)
11. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124. IEEE Computer Society, Los Alamitos (2006)
12. Kupferman, O., Lustig, Y., Vardi, M.: On locally checkable properties. In: Hermann, M., Voronkov, A. (eds.) LPAR 2006. LNCS (LNAI), vol. 4246, pp. 302–316. Springer, Heidelberg (2006)
13. Kupferman, O., Vardi, M.Y.: Model checking of safety properties. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 172–183. Springer, Heidelberg (1999)
14. Kupferman, O., Vardi, M.Y.: Safraless decision procedures. In: FOCS, pp. 531–542. IEEE, Los Alamitos (2005)
15. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
16. Müller, S.M., Paul, W.J.: Computer architecture: complexity and correctness. Springer, Heidelberg (2000)
17. Pnueli, A., Rosner, R.: On the synthesis of an asynchronous reactive module. In: Ronchi, S. D., Ausiello, G., Dezani-Ciancaglini, M. (eds.) ICALP 1989. LNCS, vol. 372, pp. 652–671. Springer, Heidelberg (1989)
18. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
19. Schneider, K., Logothetis, G.: Abstraction of systems with counters for symbolic model checking. In: Mutz, M., Lange, N. (eds.) Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen, Braunschweig, Germany, pp. 31–40. Shaker, Aachen (1999)
20. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, pp. 77–84. IEEE Computer Society Press, Los Alamitos (2009)
21. Somenzi, F.: CUDD: CU decision diagram package, release 2.4.2 (2009)
22. Wegener, I.: Branching Programs and Binary Decision Diagrams. SIAM, Philadelphia (2000)

Measuring and Synthesizing Systems in Probabilistic Environments*

Krishnendu Chatterjee[1], Thomas A. Henzinger[1,2], Barbara Jobstmann[3], and Rohit Singh[4]

[1] IST Austria
[2] EPFL, Switzerland
[3] CNRS/Verimag, France
[4] IIT Bombay, India

Abstract. Often one has a preference order among the different systems that satisfy a given specification. Under a probabilistic assumption about the possible inputs, such a preference order is naturally expressed by a weighted automaton, which assigns to each word a value, such that a system is preferred if it generates a higher expected value. We solve the following optimal-synthesis problem: given an omega-regular specification, a Markov chain that describes the distribution of inputs, and a weighted automaton that measures how well a system satisfies the given specification under the given input assumption, synthesize a system that optimizes the measured value.

For safety specifications and measures that are defined by mean-payoff automata, the optimal-synthesis problem amounts to finding a strategy in a Markov decision process (MDP) that is optimal for a long-run average reward objective, which can be done in polynomial time. For general omega-regular specifications, the solution rests on a new, polynomial-time algorithm for computing optimal strategies in MDPs with mean-payoff parity objectives. We present some experimental results showing optimal systems that were automatically generated in this way.

1 Introduction

Quantitative reasoning is traditionally used to measure quantitative properties of systems, such as performance or reliability (cf. [1, 4, 30, 33]). More recently, quantitative reasoning has been shown useful also in the classically Boolean contexts of verification (where we ask if a given system satisfies a given specification) and synthesis (where a system is automatically derived from a specification) [6, 31]. In particular, by augmenting a Boolean specifications with a quantitative specifications, we can measure how "well" a system satisfies the specification. For example, among systems that respond to requests, we may prefer one system over another if it responds quicker, or it responds to more requests, or it issues fewer unrequested responses, etc. In synthesis, we can use such measures to guide the synthesis process towards deriving a system that is, in the desired sense, "optimal" among all systems that satisfy the specification [6].

* This research was supported by the European Union project COMBEST and the European Network of Excellence ArtistDesign.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 380–395, 2010.

There are many ways to define a quantitative measure that captures the "goodness" of a system with respect to the Boolean specification, and particular measures can be quite different, but there are two questions every such measure has to answer: (1) how to assign a quantitative value to one particular behavior of a system (measure along a behavior) and (2) how to aggregate the quantitative values that are assigned to the possible behaviors of the system (measure across behaviors). Recall the response property. Suppose there is a sequence of requests along a behavior and we are interested primarily in response time, i.e., the quicker the system responds, the better. As measure (1) along a particular behavior, we may be interested in an average or the supremum (i.e., worst case) of all response times, or in any other function that aggregates all response times along a behavior into a single real value. The choice of measure (2) across behaviors is independent: we may be interested in an average of all values assigned to individual behaviors, or in the supremum, or again, in some other function. In this way, we can measure the average (across behaviors) of average (along a behavior) response times, or the average of worst-case response times, or the worst case of average response times, or the worst case of worst-case response times, etc. Note that these are the same two choices that appear in weighted automata and max-plus algebras (cf. [17, 25, 28]).

In previous work, we studied various measures (1) along a behavior. In particular, lexicographically ordered tuples of averages [6] and ratios [7] are of natural interest in certain contexts. Alur et al. [2] consider an automaton model with a quantitative measure (1) that is defined with respect to certain accumulation points along a behavior. However, in all of these cases, for measure (2) only the worst case (i.e., supremum) is considered. This comes natural as an extension of Boolean thinking, where a system fails to satisfy a property if even a single behavior violates the property. But in this way, we cannot distinguish between two systems that have the same worst cases across behaviors, but in one system almost all possible behaviors exhibit the worst case, while in the other only very few behaviors do so. In contrast, in performance evaluation one usually considers the average case across different behaviors.

For instance, consider a resource controller for two clients. Clients send requests, and the controller grants the resource to one of them at a time. Suppose we prefer, again, systems where requests are granted "as quickly as possible." Every controller that avoids simultaneous grants will have a behavior along which at least one grant is delayed by one step, namely, the behavior along which both clients continuously send requests. The best the controller can do is to alternate between the clients. Now, if systems are measured with respect to the worst-case behaviors, then a controller that always alternates between both clients, independent of the actual requests, is as good as a controller that tries to grant all requests immediately and only alternates when both clients request the resource at the same time. Clearly, if we wish to synthesize the preferred controller, we need to apply an average-case measure across behaviors.

In this paper, we present a measure (2) that averages across all possible behaviors of a system and solve the corresponding synthesis problem to derive an optimal system. In synthesis, the different possible behaviors of a system are caused by different input sequences. Therefore, in order to take a meaningful average across different behaviors, we need to assume a probability distribution over the possible input sequences. For

example, if on input 0 a system has response time r_0, and on input 1 response time r_1, and input 0 is twice as likely as input 1, then the average response time is $(2r_0 + r_1)/3$.

The resulting synthesis problem is as follows: given a Boolean specification φ, a probabilistic input assumption μ, and a measure that assigns to each system M a value $\mathcal{V}^{\varphi}_{\mu}(M)$ of how "well" M satisfies φ under μ, construct a system M such that $\mathcal{V}^{\varphi}_{\mu}(M) \geq \mathcal{V}^{\varphi}_{\mu}(M')$ for all M'. We solve this problem for qualitative specifications that are given as ω-automata, input assumptions that are given as finite Markov chains, and a quantitative specification given as mean-payoff automata which defines a quantitative language by assigning values to behaviors. From the above three inputs we derive a measure that captures (1) an average along system behaviors as well as (2) an average across system behaviors; and thus we obtain a measure that induces a value for each system.

To our knowledge this is the first solution of a synthesis problem for an average-case measure across system behaviors. Technically the solution rests on a new, polynomial-time algorithm for computing optimal strategies in MDPs with mean-payoff parity objectives. In contrast to MDPs with mean-payoff objectives, where pure memoryless optimal strategies exist, optimal strategies for mean-payoff parity objectives in MDPs require infinite memory. It follows from our result that the infinite memory can be captured with a counter, and with this insight we develop the polynomial time algorithm for solving MDPs with mean-payoff parity objectives

Related works. Many formalisms for quantitative specifications have been considered in the literature [2, 8–11, 19, 20, 23, 24, 32]; most of these works (other than [2, 11, 19]) do not consider mean-payoff specifications and none of these works focus on how quantitative specifications can be used to obtain better implementations for the synthesis problem. Furthermore, several notions of metrics for probabilistic systems and games have been proposed in the literature [21, 22]; these metrics provide a measure that indicates how close are two systems with respect to all temporal properties expressible in a logic; whereas our work uses quantitative specification to compare systems wrt the property of interest. Similar in spirit but based on a completely different technique, is the work by Niebert et al. [34], who group behaviors into good and bad with respect to satisfying a given LTL specification and use a CTL*-like analysis specification to quantify over the good and bad behaviors. This measure of logical properties was used by Katz and Peled [31] to guide genetic algorithms to discover counterexamples and corrections for distributed protocols. Control and synthesis in the presence of uncertainty has been considered in several works such as [3, 5, 16]: in all these works, the framework consists of MDPs to model nondeterministic and probabilistic behavior, and the specification is a Boolean specification. In contrast to these works where the probabilistic choice represent uncertainty, in our work the probabilistic choice represent a model for the environment assumption on the input sequences that allows us to consider the system as a whole. Moreover, we consider quantitative objectives. MDPs with mean-payoff objectives are well studied. The books [26, 36] present a detailed analysis of this topic. We present a polynomial-time solution to a more general condition: the Boolean combination of mean-payoff and parity condition on MDPs.

2 Preliminaries

Alphabet, words, and languages. An *alphabet* Σ consists of a finite set of *letters* $\sigma \in \Sigma$. A *word* w over Σ is either a *finite* or *infinite* sequence of letters, i.e., $w \in \Sigma^* \cup \Sigma^\omega$. Given a word $w \in \Sigma^\omega$, we denote by w_i the letter at position i of w and by w^i the prefix of w of length i, i.e., $w^i = w_1 w_2 \dots w_i$. We denote by $|w|$ the length of the word w, i.e., $|w^i| = i$ and $|w| = \infty$, if w is infinite. A *qualitative language* L is a subset of Σ^ω. A *quantitative language* L [11] is a mapping from the set of words to the set of reals, i.e., $L : \Sigma^\omega \to \mathbb{R}$. Note that the characteristic function of a qualitative language L is a quantitative language mapping words to 0 and 1. Given a qualitative language L, we use L also to denote its characteristic function.

Automata with parity, safety, and mean-payoff objective. An *(finite-state) automaton* is a tuple $A = (\Sigma, Q, q_0, \Delta)$, where Σ is a *alphabet*, Q is a (finite) set of *states*, $q_0 \in Q$ is an *initial state*, and $\Delta : Q \times \Sigma \rightharpoonup Q$[1] is a *transition function* that maps a state and a letter to a successor state. The *run of A on a word $w = w_0 w_1 \dots$* is a sequence of states $\rho = \rho_0 \rho_1 \dots$ such that (i) $\rho_0 = q_0$ and (ii) for all $0 \leq i \leq |w|$, $\Delta(\rho_i, w_i) = \rho_{i+1}$). A *parity automaton* is a tuple $A = ((\Sigma, Q, q_0, \Delta), p)$, where (Σ, Q, q_0, Δ) is a finite-state automaton and $p : Q \to \{0, 1, \dots, d\}$ is a *priority function* that maps every state to a natural number in $[0, d]$ called *priority*. A parity automaton A *accepts a word* w if the least priority of all states occurring infinitely often in the run ρ of A on w is even, i.e., $\min_{q \in \mathrm{Inf}(\rho)} p(q)$ is even, where $\mathrm{Inf}(\rho) = \{q \mid \forall i \exists j > i \, \rho_j = q\}$. The *language* of A denoted by L_A is the set of all words accepted by A. A *safety automaton* is a parity automaton with only priorities 0 and 1, and no transitions from priority-1 to priority-0 states. A *mean-payoff automaton* is a tuple $A = ((\Sigma, Q, q_0, \Delta), r)$, where (Σ, Q, q_0, Δ) is a finite-state automaton and $r : Q \times \Sigma \to \mathbb{N}$ is a *reward function* that associates to each transition of the automaton a *reward* $v \in \mathbb{N}$. A mean-payoff automaton assigns to each word w the long-run average of the rewards, i.e., for a word w let ρ be the run of A on w, then we have $L_A(w) = \frac{1}{n} \cdot \sum_{i=1}^{n} r(\rho_i, w_i)$, if w is finite, and $L_A(w) = \liminf_{n \to \infty} L_A(w^n)$ otherwise. Note that L_A is a function assigning values to words.

State machines and specifications. A *(finite-)state machine* (or *system*) with *input signals* I and *output signals* O is a tuple $M = (Q, q_0, \Delta, \lambda)$, where $(\Sigma_I, Q, q_0, \Delta)$ with $\Sigma_I = 2^I$ is a (finite-state) automaton and $\lambda : Q \times \Sigma_I \to \Sigma_0$ with $\Sigma_I = 2^I$ and $\Sigma_O = 2^O$ is a *labeling function* that maps every transition in Δ to an element in Σ_O. The sets Σ_I and Σ_O are called the *input and the output alphabet of* M, respectively. We denote the joint alphabet $2^{I \cup O}$ by Σ. Given an input word $w \in \Sigma_I^* \cup \Sigma_I^\omega$, let ρ by the run of M on w, the *outcome* of M on w, denoted by $\mathcal{O}_M(w)$, is the word $v \in \Sigma^* \cup \Sigma^\omega$ s.t. for all $0 \leq i \leq |w|$, $v_i = w_i \cup \lambda(\rho_i, w_i)$. So, $\mathcal{O}_M$ is the function mapping input words to outcomes. The *language* of M, denoted L_M, is the set of all outcomes of M.

We analyze state machines with respect to qualitative and quantitative specifications. *Qualitative specifications* are qualitative languages, i.e., subsets of Σ^ω or equivalently functions mapping words to 0 and 1. We consider ω-regular specifications given as safety or parity automata. Given a safety or parity automaton A and a state machine M, we say M *satisfies* L_A (written $M \models L_A$) if $L_M \subseteq L_A$ or equivalently $\forall w \in \Sigma_I^\omega$:

[1] Note that our automata are deterministic and complete to simplify the presentation.

$L_A(\mathcal{O}_M(w)) = 1$. *A quantitative specification* is given by a quantitative language L, i.e., a function that assigns values to words. Given a state machine M, we use function composition to relate L and M, i.e., $L \circ \mathcal{O}_M$ is mapping every input word w of M to the value assigned by L to the outcome of M on w. We consider quantitative specifications given by Mean-payoff automata.

Markov chains and Markov Decision Processes (MDP). A *probability distribution* over a finite set S is a function $d : S \to [0, 1]$ such that $\sum_{q \in Q} d(q) = 1$. We denote the set of all probabilistic distributions over S by $\mathcal{D}(S)$. A *Markov Decision Process (MDP)* $G = (S, s_0, E, S_1, S_P, \delta)$ consists of a finite set of *states* S, an *initial state* $s_0 \in S$, a set of *edges* $E \subseteq S \times S$, a partition (S_1, S_P) of the set S, and a probabilistic transition function δ: $S_P \to \mathcal{D}(S)$. The states in S_1 are the *player*-1 states, where player 1 decides the successor state; and the states in S_P are the *probabilistic* states, where the successor state is chosen according to the probabilistic transition function δ. So, we can view an MDP as a game between two players: player 1 and a *random player* that plays according to δ. We assume that for $s \in S_P$ and $t \in S$, we have $(s, t) \in E$ iff $\delta(s)(t) > 0$, and we often write $\delta(s, t)$ for $\delta(s)(t)$. For technical convenience we assume that every state has at least one outgoing edge. For a state $s \in S$, we write $E(s)$ to denote the set $\{t \in S \mid (s, t) \in E\}$ of possible successors. If the set $S_1 = \emptyset$, then G is called a *Markov Chain* and we omit the partition (S_1, S_P) from the definition. A *Σ-labeled MDP* (G, λ) is an MDP G with a labeling function $\lambda : S \to \Sigma$ assigning to each state of G a letter from Σ. We assume that labeled MDPs are deterministic and complete, i.e., (i) $\forall (s, s'), (s, s'') \in E$, $\lambda(s') = \lambda(s'') \to s' = s''$ holds, and (ii) $\forall s \in S, \sigma \in \Sigma$, $\exists s' \in S$ s.t. $(s, s') \in E$ and $\lambda(s') = \sigma$.

Plays and strategies. An infinite path, or a *play*, of the MDP G is an infinite sequence $\omega = s_0 s_1 s_2 \ldots$ of states such that $(s_k, s_{k+1}) \in E$ for all $k \in \mathbb{N}$. Note that we use ω for infinite sequences of states (plays) and v for finite sequences of states. We write Ω for the set of all plays, and for a state $s \in S$, we write $\Omega_s \subseteq \Omega$ for the set of plays starting at s. A *strategy* for player 1 is a function π: $S^* S_1 \to \mathcal{D}(S)$ that assigns a probability distribution to all finite sequences $v \in S^* S_1$ of states ending in a player-1 state. Player 1 follows π, if she make all her moves according to the distributions provided by π. A strategy must prescribe only available moves, i.e., for all $v \in S^*$, $s \in S_1$, and $t \in S$, if $\pi(vs)(t) > 0$, then $(s, t) \in E$. We denote by Π the set of all strategies for player 1. Once a starting state $s \in S$ and a strategy $\pi \in \Pi$ is fixed, the outcome of the game is a random walk ω_s^π for which the probabilities of every *event* $\mathcal{A} \subseteq \Omega$, which is a measurable set of plays, are uniquely defined. For a state $s \in S$ and an event $\mathcal{A} \subseteq \Omega$, we write $\mu_s^\pi(\mathcal{A})$ for the probability that a play belongs to $\mathcal{A}$ if the game starts from the state s and player 1 follow the strategy π, respectively. For a measurable function $f : \Omega \to \mathbb{R}$ we denote by $\mathbb{E}_s^\pi[f]$ the *expectation* of the function f under the probability measure $\mu_s^\pi(\cdot)$. Strategies that do not use randomization are called pure. A player-1 strategy π is *pure* if for all $v \in S^*$ and $s \in S_1$, there is a state $t \in S$ such that $\pi(vs)(t) = 1$. A *memoryless* player-1 strategy depends only on the current state, i.e., for all $v, v' \in S^*$ and for all $s \in S_1$ we have $\pi(vs) = \pi(v's)$. A memoryless strategy can be represented as a function π: $S_1 \to \mathcal{D}(S)$. A *pure memoryless strategy* is a strategy that is both pure and memoryless and can be represented as π: $S_1 \to S$.

Quantitative objectives. A quantitative objective is given by a measurable function $f : \Omega \to \mathbb{R}$. We consider several objectives based on priority and reward functions. Given a priority function $p : S \to \{0, 1, \ldots, d\}$, we defined the set of plays satisfying the parity objective as $\Omega_p = \{\omega \in \Omega \mid \min\big(p(\mathrm{Inf}(\omega))\big) \text{ is even}\}$. A *Parity objective* parity_p is the characteristic function of Ω_p. Given a reward function $r : S \to \mathbb{N} \cup \{\bot\}$, the *mean-payoff objective* mean_r for a play $\omega = s_1 s_2 s_3 \ldots$ is defined as $\mathsf{mean}_r(\omega) = \liminf_{n\to\infty} \frac{1}{n} \cdot \sum_{i=1}^{n} r(s_i)$, if for all $i > 0 : r(s_i) \neq \bot$, otherwise $\mathsf{mean}_r(\omega) = \bot$. Given a priority function p and a reward function r the *mean-payoff parity objective* $\mathsf{mp}_{p,r}$ assigns the long-run average of the rewards if the parity objective is satisfied; otherwise it assigns $\bot$. Formally, for a play ω we have $\mathsf{mp}_{p,r}(\omega) = \begin{cases} \mathsf{mean}_r(\omega) & \text{if } \mathsf{parity}_p(\omega) = 1, \\ \bot & \text{otherwise.} \end{cases}$ For a reward function $r : S \to \mathbb{R}$ the *max objective* max_r assigns to a play the maximum reward that appears in the play. Note that since S is finite, the number of different rewards appearing in a play is finite and hence the maximum is defined. Formally, for a play $\omega = s_1 s_2 s_3 \ldots$ we have $\mathsf{max}_r(\omega) = \max\langle r(s_i)\rangle_{i\geq 0}$.

Values and optimal strategies. Given an MDP G, the *value* function V_G for an objective f is the function from the state space S to the set $\mathbb{R}$ of reals. For all states $s \in S$, let $\mathsf{V}_G(f)(s) = \sup_{\pi\in\Pi} \mathbb{E}_s^{\pi}[f]$. In other words, the value $\mathsf{V}_G(f)(s)$ is the maximal expectation with which player 1 can achieve her objective f from state s. A strategy π is *optimal* from state s for objective f if $\mathsf{V}_G(f)(s) = \mathbb{E}_s^{\pi}[f]$. For parity objectives, mean-payoff objectives, and max objectives pure memoryless optimal strategies exist in MDPs.

Almost-sure winning states. Given an MDP G and a priority function p, we denote by $W_G(\mathsf{parity}_p) = \{s \in S \mid \mathsf{V}_G(\mathsf{parity}_p)(s) = 1\}$, the set of states with value 1. These states are called the *almost-sure* winning states and an optimal strategy from the almost-sure winning states is called a almost-sure winning strategy. The set $W_G(\mathsf{parity}_p)$ for an MDP G with priority function p can be computed in $O(d \cdot n^{\frac{3}{2}})$ time, where n is the size of the MDP G and d is the number of priorities [14, 15]. For states in $S \setminus W_G(\mathsf{parity}_p)$ the parity objective is falsified with positive probability for all strategies, which implies that for all states in $S \setminus W_G(\mathsf{parity}_p)$ the value is less than 1 (i.e., $\mathsf{V}_G(\mathsf{parity}_p)(s) < 1$).

3 Measuring Systems

In this section, we start with an example to explain the problem and introduce our measure. Then, we define the measure formally and show finally, how to compute the value of a system with respect to the given measure.

Example 1. Recall the example from the introduction, where we consider a resource controller for two clients. Client i requests the resource by setting its request signal r_i. The resource is granted to Client i by raising the grant signal g_i. We require that the controller guarantees mutually exclusive access and that it is fair, i.e., a requesting client eventually gets access to the resource. Assume we prefer controllers that respond quickly. Fig. 1 shows a specification that rewards a quick response to request r_i. The

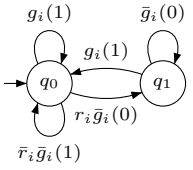


Fig. 1. Automaton A_i

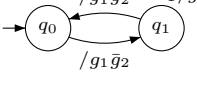


Fig. 2. System M_1

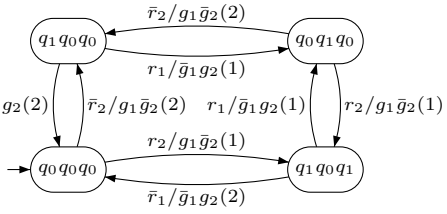


Fig. 3. Product of System M_1 with Specification A_1 and A_2

specification is given as a Mean-payoff automaton that measures the average delay between a request r_i and a corresponding grant g_i. Transitions are labeled with a conjunction of literals[2] and a reward in parentheses. In particular, whenever a request is granted the reward is 1, while a delay of the grant results in reward 0. The automaton assigns to each word in $(2^{\{r_i,g_i\}})^\omega$ the average reward. E.g., the value of the word $(r_i\bar{g}_i\ r_ig_i)^\omega$ is $(0+1)/2 = 1/2$. We can take two copies of this specification, one for each client, and assign to each word in $(2^{\{r_1,r_2,g_1,g_2\}})^\omega$ the sum of the average rewards. E.g., the word $(r_1\bar{r}_2\bar{g}_1g_2\ r_1\bar{r}_2g_1\bar{g}_2)^\omega$ gets an average reward of $1/2$ with respect to the first client and reward 1 with respect to the second client, which sums up to a total reward of $3/2$.

Consider the systems M_1 and M_2 in Fig. 2 and 4, respectively. Transitions are labeled with conjunctions of input and output literals separated by a slash. System M_1 alternates between granting the resource to Client 1 and 2. System M_2 grants the resource to Client 2, if only Client 2 is sending requests. By default it grants the resource to Client 1. If both clients request, then the controller alternates between them. Both systems are correct with respect to the functional requirements describe above: they are fair to both clients and guarantee that the resource is not accessed simultaneously.

Though, one can argue that System M_2 is better than M_1 because the delay between requests and grants is, for most input sequences, smaller than the delay in System M_1. For instance, consider the input trace $(\bar{r}_1r_2\ r_1\bar{r}_2)^\omega$. The response of System M_1 is $(g_1\bar{g}_2\ \bar{g}_1g_2)^\omega$. Looking at the product between the system M_1 and the specifications A_1 and A_2 shown in Fig. 3, we can see that this results in an average reward of 1. Similar, Fig. 5 shows the product of M_2, A_1, and A_2. System M_2 responds with $(\bar{g}_1g_2\ g_1\bar{g}_2)^\omega$ and obtains a reward of 2. Now, consider the sequence $(r_1r_2)^\omega$, which is the worst input sequence the environment can provide. In both systems, this sequences leads to a reward of 1, which is the lowest possible reward. So M_1 and M_2 cannot be distinguished with respect to their worst case behavior.

To measure a system with respect to its average behavior, we aim to average over the rewards obtained for all possible input sequences. Since we have infinite sequences, one way to average is the limit of the average over all finite prefixes. Note that this can only be done if we know the values of finite words with respect to the quantitative specification. For instance, for a finite-state machine M and a Mean-payoff automaton A, we can define the average as $\mathcal{V}_\varnothing^{L_A}(M) := \lim_{n\to\infty} \frac{1}{|\Sigma_I|^n} \sum_{w\in\Sigma_I^n} L_A(\mathcal{O}_M(w^n))$. However, if we truly want to capture the average behavior, we need to know, how often the different

[2] Note that transitions depend only on the signals that appear in their labels.

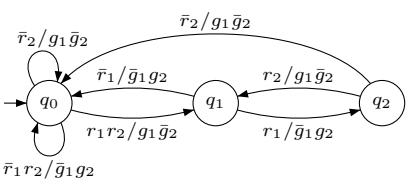


Fig. 4. System M_2 that prefers r_1

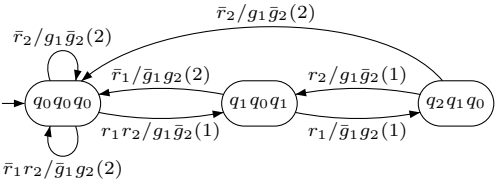


Fig. 5. Product of M_1, A_1, and A_2

parts of the system are used. This corresponds to knowing how likely the different input sequences are. The measure above assumes that all input sequences are "equally likely". In order to define measures that take the behavior of the environment into account, we use a probability measure on input words. In particular, we consider the probability space $(\Sigma_I^\omega, \mathcal{F}, \mu)$ over Σ_I^ω, where $\mathcal{F}$ is the σ-algebra generated by the cylinder sets of Σ^ω (which are the sets of infinite words sharing a common prefix) (in other words, we have the Cantor topology on Σ_I^ω) and μ is a probability measure defined on $(\Sigma^\omega, \mathcal{F})$. We use finite labeled Markov chains to define the probability measure μ.

Example 2. Recall the controller of Example 1. Assume we know that Client 1 is more likely to send requests than Client 2. We can represent such a behavior by assigning probabilities to the events in $\Sigma = 2^{\{r_1, r_2\}}$. Assume Client 1 sends requests with probability p_1 and Client 2 sends them with probability $p_2 < p_1$, independent of what has happened before. Then, we can build a labeled Markov chain with four states $S_p = \{s_0, s_1, s_2, s_3\}$ each labeled with a letter in Σ, i.e., $\lambda(s_0) = \bar{r}_1\bar{r}_2$, $\lambda(s_1) = \bar{r}_1 r_2$, $\lambda(s_2) = r_1\bar{r}_2$, and $\lambda(s_3) = r_1 r_2$., and the following transition probabilities: (i) $\delta(s_i)(s_0) = (1-p_1)\cdot(1-p_2)$, (ii) $\delta(s_i)(s_1) = (1-p_1)\cdot p_2$, (iii) $\delta(s_i)(s_2) = p_1\cdot(1-p_2)$, and (iv) $\delta(s_i)(s_3) = p_1 \cdot p_2$, for all $i \in \{0,1,2,3\}$.

Once we have a probability measure μ on the input sequences and the associated expectation measure $\mathbb{E}$, we can define a satisfaction relation between systems and specifications and a measure for a system with respect to a qualitative and a quantitative specification.

Definition 1 (Satisfaction). *Given a state machine M with input alphabet Σ_I, a qualitative specification φ, and a probability measure μ on $(\Sigma_I^\omega, \mathcal{F})$, we say that M* satisfies *φ* under *μ (written $M \models_\mu \varphi$) iff M satisfies φ with probability 1, i.e., $\mathbb{E}[\varphi \circ \mathcal{O}_M] = 1$, where $\mathbb{E}$ is the expectation measure for μ.*

Recall that we use a quantitative specification to describe how "good" a system is. Since we aim for a system that satisfies the given (qualitative) specification and is "good" in a given sense, we define the value of a machine with respect to a qualitative and a quantitative specification.

Definition 2 (Value). *Given a state machine M, a qualitative specification φ, quantitative specification ψ, and a probability measure μ on $(\Sigma_I^\omega, \mathcal{F})$, the* value of M with respect to φ and ψ under μ *is defined as the expectation of the function $\psi \circ \mathcal{O}_M$ under the probability measure μ if M* satisfies *φ* under *μ, and $\perp$ otherwise. Formally, let $\mathbb{E}$ be the expectation measure for μ, then*

$$\mathcal{V}_\mu^{\varphi\psi}(M) := \begin{cases} \mathbb{E}[\psi \circ \mathcal{O}_M] & \text{if } M \models_\mu \varphi, \\ \bot & \text{otherwise.} \end{cases}$$

If φ is the set of all words, then we write $\mathcal{V}_\mu^{\psi}(M)$. Furthermore, we say M optimizes *ψ under μ, if $\mathcal{V}_\mu^{\psi}(M) \geq \mathcal{V}_\mu^{\psi}(M')$ for all systems M'.*

We could consider here also the traditional satisfaction relation, i.e., $M \models \varphi$. We have algorithms for both notions but we focus on satisfaction under μ, since satisfaction with probability 1 is the natural correctness criterion, if we are given a probabilistic environment assumption. Note that for safety specifications the two notions coincide, because we assume that the labeled Markov chain defining the input distribution is complete.[3] For parity specifications, the results in this section would change only slightly if we replace $M \models_\mu \varphi$ by $M \models \varphi$. In particular, instead of analyzing a Markov chain with parity objective, we would have to analyze an automaton with parity objective. We discuss the the alternative synthesis algorithm in the conclusions.

Lemma 1. *Given a finite-state machine M, a safety or a parity automaton A, a mean-payoff automaton B, and a labeled Markov chain (G, λ_G) defining a probability measure μ on $(\Sigma_I^\omega, \mathcal{F})$, we can construct a Markov chain $G' = (S', s'_0, E', \delta')$, a reward function r', and a priority function p' such that*

$$\mathcal{V}_\mu^{L_A, L_B}(M) = \begin{cases} 2 \cdot \mathsf{V}_{G'}(\mathsf{mean}_{r'})(s'_0) & \text{if } A \text{ is a safety automaton,} \\ 2 \cdot \mathsf{V}_{G'}(\mathsf{mp}_{p',r'})(s'_0) & \text{otherwise.} \end{cases}$$

We first build the product of M, A, B (cf. Fig. 3). Then, G' alternates between (1) moving according to G, which means choosing an input value according to the distribution given by G, and (2) moving in $M \times A \times B$ according to the chosen input. The reward given by B for this transition is assigned to the intermediate state. The priorities are copied from A. The value $\mathcal{V}_\mu^{L_B}(M)$ is twice the expectation $\mathsf{V}_{G'}(\mathsf{mean}_{r'})(s'_0)$, since we have introduced 0-rewards in every second step. Using Lemma 1 and the fact that we can compute $\mathsf{V}_{G'}(\mathsf{mean}_{r'})(s'_0)$ and $\mathsf{V}_{G'}(\mathsf{mp}_{p',r'})(s'_0)$ in polynomial time for Markov chains [14, 26], we obtain the following results. Detailed proofs can be found in the technical report [12].

Theorem 1. *Given a finite-state machine M, a parity automaton A, a mean-payoff automaton B, and a labeled Markov chain (G, λ_G) defining a probability measure μ, we can compute the value $\mathcal{V}_\mu^{L_A, L_B}(M)$ in polynomial time. Furthermore, if (G, λ_G) defines a uniform input distribution, then $\mathcal{V}_\oslash^{L_B}(M) = \mathcal{V}_\mu^{L_B}(M)$*[4].

Example 3. Recall the two system M_1 and M_2 (Fig. 2 and 4, respectively) and the specification A (cf. Fig. 1) that rewards quick responses. The two systems are equivalent

[3] Recall that a Markov chain is complete, if in every state there is an edge for every input value. Since every edge has a positive probability, also every finite path has a positive probability and therefore a system violating a safety specification will have a value $\bot$. If the Markov chain is not complete (i.e., we are given an input distribution that assigns probability 0 to some finite input sequences), we require a simple pre-processing step that restricts our algorithms to the set of states satisfying the safety condition independent of the input assumption. This set can be computed in linear time by solving a safety game.

[4] We can show that this measure is invariant under transformations of the computation tree.

wrt the worst case behavior. Let us consider the average behavior: we build a Markov chain $G_\oslash$ that assigns $1/4$ to all events in $2^{\{r_1,r_2\}}$. To measure M_1, we take the product between $G_\oslash$ and $M_1 \times A$ (shown in Fig. 3). The product looks like the automaton in Fig. 3 with an intermediate state for each edge. This state is labeled with the reward of the edge. All transition leading to intermediate states have probability $1/2$, the other once have probability 1. So the expectation of being in a state is the same for all four main states (i.e., $1/8$) and half of it in the eight intermediate states (i.e., $1/16$). Four (intermediate) states have a reward of 1, four have a reward of 2. So we get a total reward of $4 \cdot 1/16 + 4 \cdot 2 \cdot 1/16 = 3/4$, and a system value of 1.5. This is expected when looking at Fig. 3 because each state has two inputs resulting in a reward of 2 and two inputs with reward 1. For System M_2, we obtain Markov chain similar to Fig. 5 but now the probability of the transitions corresponding to the self-loops on the initial state sum up to $3/4$. So it is more likely to state in the initial state, then to leave it. The expectation for being in the states (q_0, q_0, q_0),(q_1, q_0, q_1), and (q_2, q_1, q_0) are $2/3$, $2/9$, and $1/9$, respectively, and their expected rewards are $(2 + 2 + 2 + 1)/4 = 7/4$, $3/2$, and $3/2$, respectively. So, the total reward of System M_2 is $2/3 \cdot 7/4 + 2/9 \cdot 3/2 + 1/9 \cdot 3/2 = 1.67$, which is clearly better than the value of system M_1 for specification A.

4 Synthesizing Optimal Systems

In this section, we show how to construct a system that satisfies a qualitative specification and optimizing a quantitative specification under a given probabilistic environment. First, we reduce the problem to finding an optimal strategy in an MDP for a mean-payoff (parity) objective. Then, we show how to compute such a strategy using end components and a reduction to max objective. Finally, we provide a linear program that computes the value function of an MDP with max objective. This shows that MDPs with mean-payoff parity objective can be solved in polynomial time.

Lemma 2. *Given a safety (resp. parity) automaton A, a mean-payoff automaton B, and a labeled Markov chain (G, λ_G) defining a probability measure μ on $(\Sigma_I^\omega, \mathcal{F})$, we can construct a labeled MDP $(G', \lambda_{G'})$ with $G' = (S', s'_0, E', S'_1, S'_P, \delta')$, a reward function r', and a priority function p' such that every pure strategy π that is optimal from state s'_0 for the objective $\mathsf{mean}_{r'}$ (resp. $\mathsf{mp}_{p',r'}$) and for which $\mathbb{E}^\pi_{s'_0}(\mathsf{mean}_{r'}) \neq \bot$ (resp. $\mathbb{E}^\pi_{s'_0}(\mathsf{mp}_{p',r'}) \neq \bot$) corresponds to a state machine M that satisfies L_A under μ and optimizes L_B under μ.*

The construction of G' is very similar to the construction used in Lemma 1. Intuitively, G' alternates between mimicking a move of G and mimicking a move of $A \times B \times C$, where C is an automaton with $|\Sigma_O|$-states that pushes the output labels from transitions to states, i.e., the transition function δ_C of C is the largest transition function s.t. $\forall s, s', \sigma, \sigma' : \delta_C(s, \sigma) = \delta_C(s', \sigma') \rightarrow \sigma = \sigma'$. Priorities p' are again copied from A and rewards r' from B. The labels for $\lambda_{G'}$ are either taken from λ_G (in intermediate state) or they correspond to the transitions taken in C. Every pure strategy in G' fixes one output value for every possible input sequence. The construction of the state machine depends on the structure of the strategy. For pure memoryless strategies, the

construction is straight forward. At the end of this section, we discuss how to deal with other strategies.

The following theorem follows from Lemma 2 and the fact that MDPs with mean-payoff objective have pure memoryless optimal strategies and they can be computed in polynomial time (cf. [26]).

Theorem 2. *Given a safety automaton A, a mean-payoff automaton B, and a labeled Markov chain (G, λ_G) defining a probability measure μ, we can construct a finite-state machine M (if one exists) in polynomial time that satisfies L_A under μ and optimizes L_B under μ.*

MDPs with mean-payoff parity objectives. It follows from Lemma 2 that if the qualitative specification is a parity automaton, along with the Markov chain for probabilistic input assumption, and mean-payoff automata for quantitative specification, then the solution reduces to solving MDPs with mean-payoff parity objective. In the following we provide an algorithmic solution of MDPs with mean-payoff parity objective. We first present few basic results on MDPs.

End components of MDPs. Given an MDP $G = (S, s_0, E, S_1, S_P, \delta)$, a set $U \subseteq S$ of states is an *end component* [16, 18] if U is δ-closed (i.e., for all $s \in U \cap S_P$ we have $E(s) \subseteq U$) and the sub-game of G restricted to U (denoted $G \restriction U$) is strongly connected. We denote by $\mathcal{E}(G)$ the set of end components of an MDP G. Given any strategy (memoryless or not), with probability 1 the set of states visited infinitely often along a play is an end component. More precisely, given an MDP G, for all states $s \in S$ and all strategies $\pi \in \Pi$, we have $\mu_s^\pi(\{\omega \mid \mathrm{Inf}(\omega) \in \mathcal{E}(G)\}) = 1$ [16, 18]. Furthermore, for an end component $U \in \mathcal{E}(G)$, consider the memoryless strategy π_U that plays in any state s in $U \cap S_1$ all edges in $E(s) \cap U$ uniformly at random. In the Markov chain obtained by fixing π_U, the end component U is a closed connected recurrent set.

Lemma 3. *Given an MDP G and an end component $U \in \mathcal{E}(G)$, the strategy π_U ensures that for all states $s \in U$, we have $\mu_s^{\pi_U}(\{\omega \mid \mathrm{Inf}(\omega) = U\}) = 1$.*

It follows that the strategy π_U ensures that from any starting state s, any other state t is reached in finite time with probability 1. From Lemma 3 we can conclude that in an MDP the value for mean-payoff parity objectives can be obtained by computing values for end-components and then applying the maximal expectation to reach the values of the end components.

Lemma 4. *Consider an MDP G with state space S, a priority function p, and reward function r such that (a) G is an end-component (i.e., S is an end component) and (b) the minimum priority in S is even. Then the value for mean-payoff parity objective for all states coincide with the value for mean-payoff objective, i.e., for all states s we have $\mathsf{V}_G(\mathsf{mp}_{p,r})(s) = \mathsf{V}_G(\mathsf{mean}_r)(s)$.*

The proof idea is to take two strategies: one for the mean-payoff and one for the parity objective, and combine them to produce the optimal value for the mean-payoff parity objective, which is equal to the optimal mean-payoff value. We take an optimal pure memoryless strategy π_m for the mean-payoff objective and a pure memoryless strategy π_S for the stochastic shortest path to reach the states with the minimum priority

(which is even). Observe that (i) under the strategy π_S, from any state s we can reach the minimum (even) priority in finite time with probability 1; (ii) the mean-payoff value for all states is the same, because strategy π_U (from Lemma 3) ensures that every state can reach every other state in finite time with probability 1; and (iii) the strategy π_m ensures that for any $\varepsilon > 0$, there exists $j(\varepsilon) \in \mathbb{N}$ such that if π_m is played for any $\ell \geq j(\varepsilon)$ steps then the expected average of the rewards for ℓ steps is within ε of optimal mean-payoff value. So, we can achieve the optimal mean-payoff value and satisfies the parity objective by alternating between the two strategies, if we ensure to play π_m "long enough". Let β be the maximum absolute value of the rewards. The optimal strategy for mean-payoff objective is played in rounds, each having two stages. The strategy for round i is as follows: *(Stage 1)* First play the strategy π_S till the minimum priority state is reached. *(Stage 2)* Let $\varepsilon_i = 1/i$. If the game was in the first stage in this (i-th round) for k_i steps, then play the strategy π_m for ℓ_i steps such that $\ell_i \geq \max\{j(\varepsilon_i), i \cdot k_i \cdot \beta\}$. Then the strategy proceeds to round $i+1$. This strategy guarantees the satisfaction of the parity objective and the optimal mean-payoff value. A full proof can be found in [12].

The above lemma shows that in an end component if the minimum priority is even, then the value for mean-payoff parity and mean-payoff objective coincide if we consider the sub-game restricted to the end component.

Computing best end-component values. We first compute a set S^* such that every end component U with $\min(p(U))$ is even is a subset of S^*. We also compute a function $f^* : S^* \to \mathbb{R}^+$ that assigns to every state $s \in S^*$ the mean-payoff parity value that can be obtained by visiting only states of an end component that contains s. The computation of S^* and f^* is as follows: *(1)* S_0^* is the set of maximal end-components with priority 0 and for a state $s \in S_0^*$ the function f^* assigns the mean-payoff value when the sub-game is restricted to S_0^* (by Lemma 4 we know that if we restrict the game to the end-components, then the mean-payoff values and mean-payoff parity values coincide); *(2)* for $i \geq 0$, let S_{2i}^* be the set of maximal end components with states with priority $2i$ or more and that contains at least one state with priority $2i$, and f^* assigns the mean-payoff value of the MDP restricted to the set of end components S_{2i}^*. The set $S^* = \bigcup_{i=0}^{\lfloor d/2 \rfloor} S_{2i}^*$. This gives the values under the end-component consideration, and to compute the maximal reachability expectation we present the following reduction.

Transformation to MDPs with $\max$ objective. Given an MDP $G = (S, s_0, E, S_1, S_P, \delta)$ with a positive reward function $r : S \to \mathbb{R}^+$ and a priority function $p : S \to \{0, \ldots, d\}$, and let S^* and f^* be the output of the above procedure. We construct an MDP $\overline{G} = (\overline{S}, s_0, \overline{E}, \overline{S}_1, S_P, \delta)$ with a reward function $\overline{r}$ as follows: $\overline{S} = S \cup \widehat{S}^*$ (i.e., the set of states consists of the state space S and a copy $\widehat{S}^*$ of S^*), $\overline{E} = E \cup \{(s, \widehat{s}) \mid s \in S^* \cap S_1$ and $\widehat{s}$ is the copy of s in $\widehat{S^*}\} \cup \{(\widehat{s}, \widehat{s}) \mid \widehat{s} \in \widehat{S}^*\}$ (i.e., along with edges E, for all player 1 states s in S^* there is an edge to its copy $\widehat{s}$ in $\widehat{S}^*$, and all states in $\widehat{S}^*$ are absorbing states), $\overline{S}_1 = S_1 \cup \widehat{S}^*$, $\overline{r}(s) = 0$ for all $s \in S$ and $\overline{r}(\widehat{s}) = f^*(s)$, where $\widehat{s}$ is the copy of s. This construction ensures that $\mathsf{V}_G(\mathsf{mp}_{p,r})(s) = \mathsf{V}_{\overline{G}}(\mathsf{max}_{\overline{r}})(s)$. We refer the reader to [12] for a detailed proof.

In order to solve $\overline{G}$ with the objective $\mathsf{max}_{\overline{r}}$, we set up the following linear program, which can be solved with a standard LP solver (e.g., [29]).

Linear programming for the $\max$ **objective in** $\overline{G}$**.** The following linear program characterizes the value function $\mathsf{V}_{\overline{G}}(\mathsf{max}_{\overline{r}})$. Observe that we have already restricted ourselves to the almost-sure winning states $W_G(\mathsf{parity}_p)$, and below we assume $W_G(\mathsf{parity}_p) = S$. For all $s \in \overline{S}$ we have a variable x_s and the objective function is $\min \sum_{s \in \overline{S}} x_s$. The set of linear constraints are as follows: (1) $\forall s \in \overline{S} : x_s \geq 0$, (2) $\forall s \in \widehat{S}^* : x_s = \overline{r}(s)$, (3) $\forall s \in \overline{S}_1, (s,t) \in \overline{E} : x_s \geq x_t$, and (4) $\forall s \in \overline{S}_P : x_s = \sum_{t \in \overline{S}} \overline{\delta}(s)(t) \cdot x_t$. The correctness proof of this linear program can be found in [12].

Lemma 5. *Given a MDP with a mean-payoff parity objective, the value function for the mean-payoff parity objective can be computed in polynomial time.*

Note that the optimal strategies constructed for mean-payoff parity requires memory, but the memory requirement is captured by a counter (which can be represented by a state machine with state space $\mathbb{N}$). The optimal strategy as described in Lemma 4 plays two memoryless strategies, and each strategy is played a number of steps which can be stored in a counter. Furthermore, we can show that the decision problem, whether there exists an optimal pure memoryless strategy is NP-complete; the upper bound follows from Theorem 1; the lower bound follows from a reduction of the directed subgraph homeomorphism problem [27]. Lemma 2 and Lemma 5 yield the following theorem.

Theorem 3. *Given a Parity specification A, a Mean-payoff specification B, and a labeled Markov chain (G, λ) defining a probability measure μ on $(\Sigma_I^\omega, \mathcal{F})$, we can construct a state machine M (if one exists) in polynomial time that satisfies L_A under μ and optimizes L_B under μ.*

5 Experimental Results

The aim of this section is to show which types of systems, we can construct using qualitative and quantitative specifications under probabilistic environment assumptions. We have implemented the approach for specifications consisting of a safety automaton A and a mean-payoff automaton B, and where the assumption μ is given as a set of probability distributions d_s over input letters for each state s of B. Our implementation is in Scala [35]. It takes automata in GOAL-format [37] as input and first builds the product of A and B. Then, it construct the corresponding MDP G and computes an optimal pure memoryless strategy using policy iteration for multi-chain MDPs [26]. Finally, if the value of the strategy is different from $\bot$, then it converts the strategy to a finite-state machine M which satisfies L_A (under μ) and is optimal for B under μ.

Priority-driven controller. In our first experiment, we took as the quantitative specification B the product of the specifications A_1 and A_2 from Example 1 (Fig. 1), where we sum the weights on the edges. The qualitative specification is a safety automaton A ensuring mutually exclusive grants. We assumed the constant probabilities $P(\{r_1 = 1\}) = 0.4$ and $P(\{r_2 = 1\}) = 0.3$ for the events $r_1 = 1$ and $r_2 = 1$, respectively. The optimal machine[5] constructed by the tool is shown in Fig. 6. This system behaves like a priority-driven scheduler, which always grants the resource to the

[5] State q_0 and q_1 are simulation equivalent but our tool does not minimize state machines yet.

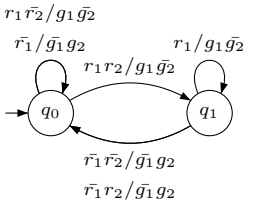


Fig. 6. Optimal machine

Table 1. Results for n-client without response constraints

n	$A \times B$	G	M	Value
2	4	13	2	1.854
3	8	35	4	2.368
4	16	97	8	2.520
5	32	275	16	2.534
6	64	793	32	2.534

Table 2. Results for n-client with response constraints

n	$A \times B$	G	M	Value
2	3	11	3	1.850
3	34	156	16	2.329
4	125	557	125	2.366

client that is more likely to send requests, if she is requesting it, otherwise the resource is granted to the other client. This is optimal because client 1 is more likely to send requests and so missing a request from client 2 is better than missing a request from 1.

Fair controller. In the second experiment, we added response constraints to the safety specification. The constraints are given as safety automata that require that every request is granted within two steps. We added one automaton C_i for each client i and the final qualitative specification was $A \times C_1 \times C_2$. The optimal machine the tool constructs is System M_2 of Example 1 (Fig. 4). System M_2 follows the request sent, if only a single request is send. If both clients request simultaneously, it alternates between g_1 and g_2. If none of the clients is requesting it grants g_1. Recall that system M_1 and M_2 from Example 1 exhibit the same worst-case behavior, so a synthesis approach based on optimizing the worst-case behavior would not be able to construct M_2.

General controllers. We reran both experiments for several clients. Again, the quantitative specification was the product of A_i's. We used a skewed probability distribution with $P(\{r_n = 1\}) = 0.3$ and $P(\{r_i = 1\}) = P(\{r_{i+1} = 1\}) + 0.1$ for $1 \leq i \leq 6$ and the qualitative specification required mutual exclusion. Table 1 shows the number of clients (n), the size of the specification ($A \times B$), the size of the corresponding MDP (G), and the size of the resulting machine (M) and the optimal value (Value). The runs took between least than a second to a couple of minutes. The systems generated as a result of this experiment have an intrinsic priority to granting requests in order of probabilities from largest to smallest. Table 2 shows the results when adding response constraints that require that every request has to be granted within the next n steps. This experiment leads to quite intelligent systems which prioritize with the most probable input request but slowly the priority shifts to the next request variable cyclically resulting into servicing any request in n steps when there are n clients. Note that these systems are (as expected) quite a bit larger than the corresponding priority-driven controllers.

6 Conclusions and Future Work

In this paper we showed how to measure and synthesize systems under probabilistic environment assumptions wrt qualitative and quantitative specifications. We considered the satisfaction of the qualitative specification with probability 1 ($M \models_\mu \varphi$). Alternatively, we could have considered the satisfaction of the qualitative specification with certainty ($M \models \varphi$). For safety specification the two notions coincide, however, they

are different for parity specification. The notion of satisfaction of the parity specification with certainty and optimizing the mean-payoff specification can be obtained similar to the solution of mean-payoff parity games [13] by replacing the solution of mean-payoff games by solution of MDPs with mean-payoff objectives. However, since solving MDPs with parity specification for certainty is equivalent to solving two-player parity games, and no polynomial time algorithm is known for parity games, the algorithmic solution for the satisfaction of the qualitative specification with certainty is computationally expensive as compared to the polynomial time algorithm for MDPs with mean-payoff parity objectives. Moreover, under probabilistic assumption satisfaction with probability 1 is the natural notion.

In our future work, we will implement our algorithm for MDPs with mean-payoff parity conditions and develop a tool for synthesizing systems in probabilistic environments with ω-regular specifications. In the course of developing this tool, it will be interesting to study subclasses of specifications for which we can construct finite-state systems (i.e., systems without counters). We will also explore the use of a logical framework to express quantitative properties to simplify stating quantitative specifications.

References

1. de Alfaro, L.: Temporal logics for the specification of performance and reliability. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, pp. 165–176. Springer, Heidelberg (1997)
2. Alur, R., Degorre, A., Maler, O., Weiss, G.: On omega-languages defined by mean-payoff conditions. In: de Alfaro, L. (ed.) FOSSACS 2009. LNCS, vol. 5504, pp. 333–347. Springer, Heidelberg (2009)
3. Baier, C., Größer, M., Leucker, M., Bollig, B., Ciesinski, F.: Controller synthesis for probabilistic systems. In: IFIP TCS, pp. 493–506 (2004)
4. Baier, C., Katoen, J.-P.: Principles of Model Checking. Representation and Mind Series. The MIT Press, Cambridge (2008)
5. 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)
6. Bloem, R., Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)
7. Bloem, R., Greimel, K., Henzinger, T.A., Jobstmann, B.: Synthesizing robust systems. In: FMCAD'09 (2009)
8. Chakrabarti, A., Chatterjee, K., Henzinger, T.A., Kupferman, O., Majumdar, R.: Verifying quantitative properties using bound functions. In: Borrione, D., Paul, W. (eds.) CHARME 2005. LNCS, vol. 3725, pp. 50–64. Springer, Heidelberg (2005)
9. Chakrabarti, A., de Alfaro, L., Henzinger, T.A., Stoelinga, M.: Resource interfaces. In: Alur, R., Lee, I. (eds.) EMSOFT 2003. LNCS, vol. 2855, pp. 117–133. Springer, Heidelberg (2003)
10. Chatterjee, K., de Alfaro, L., Faella, M., Henzinger, T.A., Majumdar, R., Stoelinga, M.: Compositional quantitative reasoning. In: QEST, pp. 179–188 (2006)
11. Chatterjee, K., Doyen, L., Henzinger, T.A.: Quantitative languages. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 385–400. Springer, Heidelberg (2008)
12. Chatterjee, K., Henzinger, T., Jobstmann, B., Singh, R.: Measuring and synthesizing systems in probabilistic environments. In: CoRR, arXiv:1004.0739 (2010)

13. Chatterjee, K., Henzinger, T.A., Jurdzinski, M.: Mean-payoff parity games. In: LICS, pp. 178–187 (2005)
14. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Simple stochastic parity games. In: Baaz, M., Makowsky, J.A. (eds.) CSL 2003. LNCS, vol. 2803, pp. 100–113. Springer, Heidelberg (2003)
15. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: SODA'04, pp. 121–130. SIAM, Philadelphia (2004)
16. Courcoubetis, C., Yannakakis, M.: Markov decision processes and regular events. In: Paterson, M. (ed.) ICALP 1990. LNCS, vol. 443, pp. 336–349. Springer, Heidelberg (1990)
17. Cuninghame-Green, R.A.: Minimax algebra. Lecture Notes in Economics and Mathematical Systems, vol. 166. Springer, Heidelberg (1979)
18. de Alfaro, L.: Formal Verification of Probabilistic Systems. PhD thesis, Stanford University (1997)
19. de Alfaro, L.: Stochastic transition systems. In: Sangiorgi, D., de Simone, R. (eds.) CONCUR 1998. LNCS, vol. 1466, pp. 423–438. Springer, Heidelberg (1998)
20. de Alfaro, L., Henzinger, T.A., Majumdar, R.: Discounting the future in systems theory. In: ICALP'03 (2003)
21. de Alfaro, L., Majumdar, R., Raman, V., Stoelinga, M.: Game relations and metrics. In: LICS, pp. 99–108. IEEE Computer Society Press, Los Alamitos (2007)
22. Desharnais, J., Gupta, V., Jagadeesan, R., Panangaden, P.: Metrics for labelled markov systems. In: Baeten, J.C.M., Mauw, S. (eds.) CONCUR 1999. LNCS, vol. 1664, pp. 258–273. Springer, Heidelberg (1999)
23. Droste, M., Gastin, P.: Weighted automata and weighted logics. Theoretical Computer Science 380, 69–86 (2007)
24. Droste, M., Kuich, W., Rahonis, G.: Multi-valued MSO logics over words and trees. Fundamenta Informaticae 84, 305–327 (2008)
25. Droste, M., Kuich, W., Vogler, H.: Handbook of Weighted Automata. Springer Publishing Company, Incorporated, Heidelberg (2009)
26. Filar, J., Vrieze, K.: Competitive Markov Decision Processes. Springer, Heidelberg (1996)
27. Fortune, S., Hopcroft, J.E., Wyllie, J.: The directed subgraph homeomorphism problem. Theor. Comput. Sci., 111–121 (1980)
28. Gaubert, S.: Methods and applications of (max, +) linear algebra. In: Reischuk, R., Morvan, M. (eds.) STACS 1997. LNCS, vol. 1200, pp. 261–282. Springer, Heidelberg (1997)
29. Glpk (gnu linear programming kit), http://www.gnu.org/software/glpk/
30. Haverkort, B.R.: Performance of Computer Communication Systems: A Model-Based Approach. John Wiley & Sons, Inc., New York (1998)
31. Katz, G., Peled, D.: Code mutation in verification and automatic code correction. In: TACAS 2010 (to appear, 2010)
32. Kupferman, O., Lustig, Y.: Lattice automata. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 199–213. Springer, Heidelberg (2007)
33. Kwiatkowska, M., Norman, G., Parker, D.: PRISM: Probabilistic model checking for performance and reliability analysis. ACM SIGMETRICS Perform. Evaluation Review (2009)
34. Niebert, P., Peled, D., Pnueli, A.: Discriminative model checking. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 504–516. Springer, Heidelberg (2008)
35. Odersky, M., Spoon, L., Venners, B.: Programming in Scala. Artima (2008), http://www.scala-lang.org/
36. Puterman, M.L.: Markov Decision Processes. John Wiley and Sons, Chichester (1994)
37. Tsay, Y.-K., Chen, Y.-F., Tsai, M.-H., Wu, K.-N., Chan, W.-C.: GOAL: A graphical tool for Büchi automata and temporal formulae. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 466–471. Springer, Heidelberg (2007), http://goal.im.ntu.edu.tw

Achieving Distributed Control through Model Checking

Susanne Graf[1], Doron Peled[2], and Sophie Quinton[1]

[1] VERIMAG, Centre Equation, Avenue de Vignate, 38610 Gières, France
[2] Department of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel

Abstract. We apply model checking of knowledge properties to the design of distributed controllers that enforce global constraints on concurrent systems. We calculate when processes can decide, autonomously, to take or block an action so that the global constraint will not be violated. When the separate processes cannot make this decision alone, it may be possible to temporarily coordinate several processes in order to achieve sufficient knowledge jointly and make combined decisions. Since the overhead induced by such coordinations is important, we strive to minimize their number, again using model checking. We show how this framework is applied to the design of controllers that guarantee a priority policy among transitions.

1 Introduction

Consider a concurrent system, where some global safety constraint, say of prioritizing transitions, needs to be imposed. A completely global coordinator can control this system and allow any of the maximal priority actions to progress in each state. However, the situation at hand is that of a distributed control [7,12]; controllers, one per process or set of processes, may restrict the execution of some of the transitions if their occurrence may violate the imposed constraint. Due to the distributed nature of the system, each controller has a limited view of the entire system. Each controller may keep some finite memory that is updated according to the history it can observe.

The *knowledge* of a process in any particular local state includes the properties that are common to all reachable (global) states containing it. There are several definitions for knowledge, depending on how much of the local history may be encoded in the local state. Knowledge was suggested as a tool for constructing a controller in [6,1]. There, controlling a distributed system was achieved by first precalculating the knowledge of a process. Based on its precalculated knowledge, reflecting all the possible current situations of the other processes, a *controller* for a process may decide at runtime whether an action of the controlled process can be executed without violating the imposed constraint. Sometimes, however, the process knowledge is not sufficient. Then, the joint knowledge of several processes (also called *distributed knowledge*) may be monitored using fixed controllers for sets of processes. Unfortunately, this approach causes the loss of actual concurrency among the processes that are jointly monitored.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 396–409, 2010.

Instead of permanent synchronizations via fixed process groups, we suggest in this paper a method for constructing distributed controllers that synchronize processes temporarily. We use model-checking techniques to precalculate a minimal set of synchronization points, where joint knowledge can be achieved during short coordination phases. An additional goal is synchronizing a minimal number of processes as rarely as possible. After each synchronization, the participating processes can again progress independently until a further synchronization is called for.

In [6], knowledge based controllability (termed *Kripke observability*) is studied as a basis for constructing a distributed controller. The problem there is somewhat different than ours: the goal is to make the system behave *exactly* according to a given regular language, while here we want to limit the possible choices in order to impose some given global invariant. There, if a transition is enabled by the controlled system but must be blocked according to the additional constraint, then at least one process knows that fact and is thus able to prevent its execution. This approach requires sufficient knowledge to allow any transition enabled according to a given regular specification. The construction in [1] is different: it requires that *at least one* process knows that the occurrence of *some* enabled transition preserves the correctness of the imposed constraint, hence supporting its execution. This approach preserves the correctness of the controller even when knowledge about other such transitions is limited, at the expense of restricting the choice of transitions.

The approach suggested here extends the knowledge based approach of [1]. We use a coordinator algorithm, such as the α-core [5], which achieves temporary multiprocess coordinations using asynchronous message passing. Such coordinations can be used to achieve a precalculated joint knowledge, i.e., knowledge common to several processes. Such interactions are still expensive as they incur additional overhead. Therefore, an important part of our task is to minimize the number of interactions and the number of processes involved in such interactions.

2 Preliminaries and Related Work

Definition 1 (Distributed Transition systems). *A distributed transition system $\mathcal{A}$ is a fivetuple $\langle \mathcal{P}, V, S, \iota, T \rangle$:*

- *$\mathcal{P}$ is a finite set of* processes.
- *V is a finite set of* variables*, each ranging over some finite domain. A process $p \in \mathcal{P}$ can access and change variables in V_p. Thus, $V = \cup_{p \in \mathcal{P}} V_p$. We do not require the sets V_p to be disjoint.*
- *S is the set of* global states*. Each state assigns a value to each variable in V according to its domain.*
- *$\iota \in S$ is the* initial state*.*
- *T is a finite set of* transitions*. A transition $\tau \in T$ consists of an* enabling condition *en_τ, which is a quantifier-free first order predicate, and a* state transformation *f_τ. The transitions $T_p \subseteq T$ are associated with process p. Thus, $T = \cup_{p \in \mathcal{P}} T_p$. A transition τ may belong to more than one process and $P_\tau = \{p | \tau \in T_p\}$. Both enabling condition and transformation are over the variables $\cup_{p \in P_\tau} V_p$.*

Definition 2. *A* local state $s|_p$ *of a process* $p \in \mathcal{P}$ *is the restriction of a global state* s *to the variables in* V_p*. Similarly, the* joint local state $s|_P$ *of a set of processes* $P \subseteq \mathcal{P}$ *is the restriction of a global state to the variables in* $\cup_{p \in P} V_p$*.*

For a set of states S of a transition system, we denote the set of local states of process p by $S|_p$, and, respectively, the set of joint local states for set of processes $P \in \mathcal{P}$ by $S|_P$. A transition τ is *enabled* in a state s when $s \models en_\tau$ (i.e., s satisfies en_τ). If τ is enabled in s and τ is executed, a new state $s' = f_\tau(s)$ is reached. We denote this by $s \xrightarrow{\tau} s'$.

Definition 3. *An* execution *of a distributed system* $\mathcal{A}$ *is a maximal sequence* $s_0\, s_1\, s_2 \ldots$ *such that* $s_0 = \iota$*, and for each* $i \geq 0$*,* $s_i \xrightarrow{\tau_i} s_{i+1}$ *for some* τ_i*. A global state is called* reachable *if it appears in some execution sequence.*

Definition 4. *Given a system* $\mathcal{A}$*, a set of processes* $P \subseteq \mathcal{P}$ knows *in a state* s *some property* φ *over* V*, if* $s' \models \varphi$ *for each* reachable *global state* s' *with* $s'|_P = s|_P$*. We denote this by* $s \models K_P \varphi$ *.*

When P is a singleton, we often write p for the set $\{p\}$ as in $K_p\varphi$. It is easy to see that if $s \models K_P\varphi$ and $s|_P = s'|_P$ then also $s' \models K_P\varphi$.

Definition 5. *A finite state* distributed disjunctive controller *[7,12] for a system* $\mathcal{A} = \langle \mathcal{P}, V, S, \iota, T \rangle$ *is a set of automata* $C_p = (L_p, \gamma_p, T_p^o, T_p^c, \rightarrow_p, E_p)$*, one per process* p *in* $\mathcal{P}$*, where:*

- L_p *is the set of* states *of* C_p*, i.e., its finite memory.*
- $\gamma_p \in L_p$ *is the initial state of* C_P*.*
- T_p^o *is the set of transitions* observable *by process* p*, satisfying* $T_p \subseteq T_p^o \subseteq T$*: only the execution of transitions from* T_p^o *can change the state of* C_p*.*
- T_p^c *is the set of* controllable *transitions, where* $T_p^c \subseteq T_p$*. We require consistency between processes regarding controllability: if* τ *is involved with several processes, then it is either controllable by all of them or by none of them.*
- $\rightarrow_p : L_p \times T_p \mapsto L_p$ *is the* transition function *of* C_p*.*
- $E_p : S|_p \times L_p \mapsto 2^{T_p^c}$ *is the* support function*, which in each local state returns the set of controlled transitions of process* p *that* C_p *supports (i.e., allows to proceed, when enabled).*

A controller is designed to impose some constraint $\psi \subseteq S \times T$ on a given system $\mathcal{A}$, while not introducing any new deadlock.

Definition 6. *A* controlled execution *of a distributed system* $\mathcal{A}$ *with controllers* C_p *for* $p \in \mathcal{P}$ *is defined over a set of* controlled states $G \subseteq S \times \Pi_{p \in \mathcal{P}} L_p$*. Each* controlled state $g \in G$ *contains some global state* $s \in S$*, and a state* $\rho_p \in L_p$ *for each controller* C_p*. An execution* $g_0\, g_1\, g_2 \ldots$ *is a maximal sequence of controlled states, satisfying that* g_0 *is the controlled state containing the initial states* ι *of* $\mathcal{A}$ *and* γ_p *for each* C_p*. Furthermore, for each adjacent pair of controlled states* g_i *and* g_{i+1} *there exists a transition* τ *such that the following holds:*

1. $s \xrightarrow{\tau} s'$ — *where* $s \in S$ *is the state component of the controlled state* g_i *and* $s' \in S$ *the one of* g_{i+1}.
2. $\tau \in T_p \setminus T_p^c \cup E_p(s|_p, \rho_p)$ *for at least one process* p*; that is, either* τ *is uncontrollable by* p *or* p *supports* τ *in its current local state and given the state of its controller* C_p.
3. *For the states* ρ_i *and* ρ_{i+1} *of controller* C_p *of* g_i *and* g_{i+1}*, respectively, if* $\tau \in T_p^o$*, then* $\rho_i \xrightarrow{\tau}_p \rho_{i+1}$*. Otherwise,* $\rho_i = \rho_{i+1}$*. That is,* C_p *changes its internal state when an observable transition occurs.*

We denote by $\mathcal{A}_c$ *the transformation of* $\mathcal{A}$ *that includes both* $\mathcal{A}$ *and its controllers.*

Definition 7. *A controller for* $\mathcal{A}$ *achieves a* goal $\psi \subseteq S \times T$ *if each transition* $s \xrightarrow{\tau} s'$ *(as in bullet 1. of Definition 6) satisfies that* $(s, \tau) \in \psi$.

Note that the goal of the controller is to satisfy an invariant that is not just over the states (of the original system $\mathcal{A}$), but may also include the immediate transition out of that state. When no constraints on the transitions are imposed, we can use the simpler case where $\psi \subseteq S$.

The definition of a controller allows the use of some finite memory that is updated with the execution of observable transitions. This can be useful, e.g., when constructing a controller based on knowledge with perfect recall [11]. However, a controller based on simple knowledge, as in Definition 4, does not have to exercise this capability, and L_p can thus consist of a single state. As in [1], we fix as a running example a particular property that we want to synthesize: that of enforcing some priority policy on the distributed system.

Definition 8 (Priority policy). *A* priority policy $Pr = (T, \ll)$ *for a system* $\mathcal{A}$ *is defined as a partial order relation* $\ll$ *on the set of transitions* T.

Among the transitions enabled in state s, we can identify those with *maximal priority*, i.e., enabled transitions such that for any other transition τ' enabled in s, either $\tau' \ll \tau$ or τ and τ' are incomparable. Let $\max_\tau$ be a predicate that holds in a state s, i.e., $s \models \max_\tau$, when the transition τ has a maximal priority among the transitions enabled in s.

Definition 9. *A* prioritized *execution of a system* $\mathcal{A}$ *according to a given priority policy* Pr *satisfies, in addition to the conditions of Definition 3, that when* $s_i \xrightarrow{\tau_i} s_{i+1}$*, then also* $s_i \models \max_{\tau_i}$.

The goal is then to construct a distributed controller for $\mathcal{A}$ such that, when running $\mathcal{A}$ together with its controller, only correctly prioritized executions occur. To prevent the situation where in some state an uncontrollable transition has lower priority than another enabled transition, we impose the restriction that uncontrollable transitions always have maximal priority.

Definition 10. *For each local state* $s|_p$ *of process* p*, define the following properties* k_i^p *based on the knowledge of* p *in that state.*

- $k_1^p = \bigvee_{\tau \in T_p} K_p \max_\tau$: *process p can identify a transition* τ *such that it knows that* τ *is enabled with maximal priority.*
- $k_2^p = \neg k_1^p \wedge K_p \bigvee_{q \neq p} k_1^q$: *process p does not know whether it has a transition with maximal priority, but in all the global states* s' *with* $s'|_p = s|_p$ *some other process q is in a local state where* k_1^q *holds. This allows p to remain inactive without risk of introducing a deadlock.*
- $k_3^p = \neg k_1^p \wedge \neg k_2^p$: *p does not know whether or not there is a supported transition.*

k_1^p *can be extended to sets of processes:* $k_1^P = \bigvee_{\tau \in \cup_{p \in P} T_p} K_P \max_\tau$.

Note that $k_1^p \vee k_2^p \vee k_3^p \equiv true$. When the constraint ψ imposed by the controller is different from the priority policy, the formula k_1^p needs to be changed accordingly; instead of $\max_\tau$, it must reflect the property that executing τ does not invalidate ψ. If ψ is a state property ($\psi \subseteq S$), then $\max_\tau$ can be replaced by the state predicate $wp_\tau(\psi)$ (for "weakest precondition"), which reflects the state property that holds when τ is enabled and ψ holds after its execution.

The construction in [1] checks whether $\bigvee_{p \in \mathcal{P}} k_1^p$ holds in all reachable states of the original system that are not deadlock (or termination). If so, it is sufficient that each process supports a transition when it knows that it is maximal in order to enforce the additional constraint ψ (in that case, priority) without introducing any additional deadlock. When this check fails, it was suggested to monitor and control several processes together, or to use the more expensive knowledge of perfect recall (or to use both).

3 A Synchronization Based Approach

In this paper, we suggest a new solution to the distributed control problem, which consists of synthesizing distributed controllers that allow processes to *temporarily synchronize* in order to obtain joint knowledge in those (local) states in which it is needed. The synchronization is achieved by using an algorithm like α-core [5]. This algorithm allows processes to notify, using asynchronous message passing, a set of coordinators about their wish to be involved in a joint action. We treat the synchronizations provided by the α-core, or any similar algorithm, as transitions that are joint between several participating processes. At a lower level, such synchronizations are achieved using asynchronous message passing. We assume that the correctness of the algorithm guarantees the atomic-like behavior of such coordinations, allowing us to reason at this level of abstraction.

A joint local state $s|_P$ satisfying k_1^P indicates that the set of processes P know how to act in this state by selecting some transition with maximal priority. Our construction calculates, using model checking for knowledge properties, which synchronizations are actually needed.

An exact check for the existence of a global (completely synchronized) controller can be based on game theory. Accordingly, one may present the problem as implementing a strategy for the following two player game. One player, the environment, can always choose between the enabled uncontrollable transitions,

while the other player can choose between the enabled controllable ones. The goal of the controller is that the property ψ is satisfied by the jointly selected execution. This can be solved using algorithms based on safety games [9].

Our algorithm first calculates the local states and joint local states (synchronizations) providing sufficient knowledge to guarantee that in every global state at least one process supports some transition. We refer to this set of (joint) local states as the *knowledge table* Δ for $\mathcal{A}$. We use it to transform the system into a controlled system by implementing support to transitions: when the knowledge is not available locally, we add temporary synchronization between processes, according to the entries of the knowledge table. Finally, we propose to obtain a more efficient controller by minimizing the set of coordinations.

3.1 A Set of Synchronizations Providing Sufficient Knowledge

First, we calculate the required *knowledge table* Δ. The construction of Δ is performed iteratively, starting with local states, then pairs of local states, triples etc. At each stage of the construction, Δ contains a set of (joint) local states $s|_P$ satisfying k_1^P.

Definition 11. *A set of (joint) local states Δ is an* invariant *of a system if each non-deadlock state of the system contains at least one (joint) local state from Δ.*

The first iteration includes in Δ, for all $p \in \mathcal{P}$, the singleton local states satisfying k_1^p, i.e. states in which progress of p is guaranteed. With each such local state $s|_p$ we *associate* the actual transitions τ that make k_1^p hold.

If Δ is not an invariant, we first calculate for each local state not satisfying k_1^p whether it satisfies k_2^p. Let U_p be the set of local states of process p satisfying $\neg(k_1^p \vee k_2^p)$. Now, in a second iteration, we add to Δ pairs $(s_p, s_q) \in U_p \times U_q$ for $p \neq q$ if there exists a reachable state s such that $s|_p = s_p$ and $s|_q = s_q$, and furthermore $s \models k_1^{\{p,q\}}$. Again, we associate with that entry of the table Δ the transitions τ that witness the satisfaction of $k_1^{\{p,q\}}$ for that entry. The second iteration terminates as soon as Δ is an invariant or if all such pairs of local states have been classified. In a third iteration, we consider triples of local states from $U_p \times U_q \times U_r$ such that no subtuple is in Δ, and so forth.

3.2 A Distributed Controller Imposing the Global Property

We transform now the system $\mathcal{A}$ into a controlled transition system $\mathcal{A}_c$ allowing only prioritized executions. We implement $\mathcal{A}_c$ using a set of coordinators realizing the required synchronization of Δ by an algorithm such as the α-core.

We want to achieve the joint local knowledge promised by the precalculation of Δ using synchronizations amongst the processes involved. Our construction guarantees that each time the transition associated with a tuple $(s|_{p_1} \ldots s|_{p_k})$ from Δ is executed from a state that includes these local components, the property ψ we want to impose is preserved. We transform the system $\mathcal{A}$ such that only transitions associated with entries in Δ can be executed.

If a transition τ is associated with a singleton element $s|_p$ in Δ, then the controller for p, at the local state $s|_p$, supports τ. Otherwise, τ is associated with a tuple of local states in Δ; when reaching any of these local states, the corresponding processes $p_1 \dots p_k$ try to achieve a synchronization, which consequently allows τ to execute. This is done according to the protocol of the synchronization algorithm that is used. Upon reaching the synchronization, the associated transition τ is then supported by any of its participating processes. Formally, for each transition τ associated with a tuple of local states $(s|_{p_1} \dots s|_{p_k})$, we execute a transition, enabled exactly in the joint local state with the above components, and performing the original transformation of τ.

3.3 Minimizing the Number of Coordinators

It is wasteful to include a coordination for each joint local state involving at least two processes in Δ. We now show how to minimize the number of coordinators for pairs of the form $(s|_p, r|_q)$ in Δ. The general version of this method for larger tuples is analogous. We denote by $\Delta_{p,q}$ the set of pairs of Δ made of a local state from process p and one from process q.

A naive implementation may use a coordination for every pair in Δ. Nevertheless, the large number of messages needed to implement coordination by an algorithm like α-core suggests that we minimize the number of coordinations. A completely opposite extreme would be to use a unique coordination between processes p and q. Accordingly, when process p identifies that it may have a q partner in $\Delta_{p,q}$, then coordination starts. When coordination succeeds, the joint event checks whether the local states of p and q actually appear in $\Delta_{p,q}$. If they do, it provides the appropriate behavior; otherwise, the coordination is abandoned. In this way, many (expensive) coordinations may be made just to be abandoned, not even guaranteeing eventual progress.

Consider now a set of pairs $\Gamma \subseteq \Delta_{p,q}$ such that if $(s, r), (s', r') \in \Gamma$, then $(s, r'), (s', r) \in \Gamma$ (s and s' do not have to be disjoint, and neither do r and r'). This means that Γ is a complete bipartite subgraph of $\Delta_{p,q}$. It is sufficient to generate one coordination for all the pairs in Γ. Upon success of the coordination, the precalculated table $\Delta_{p,q}$ will be consulted about which transition to allow, depending on the components $s|_p$ and $s|_q$. Thus, according to this strategy, a sufficient number of interactions is formed by finding a covering partition $\Gamma_1, \dots, \Gamma_m$ of complete bipartite subgraphs of $\Delta_{p,q}$. That is, each pair $(s|_p, r|_q) \in \Delta_{p,q}$ must be in some set Γ_i. However, the minimization problem for such a partition turns out to be in NP-Complete.

Property 1. [4] Given a bipartite graph $G = (N, E)$ and a positive integer $K \leq |E|$, finding whether there exists a set of subsets $N_1, \dots N_k$ for $k \leq K$ of complete bipartite subgraphs of G such that each edge (u, v) is in some N_i is in NP-Complete.

We use the following notation: when Γ is a set of pairs of local states, one from p and one from q, we denote by $\Gamma|_p$ and by $\Gamma|_q$ the p and the q components in

these pairs, respectively. We apply the following heuristics to calculate a (not necessarily minimal) set of complete bipartite subsets $\Gamma_i \subseteq \Delta_{p,q}$ covering $\Delta_{p,q}$. We start with a first partition $\Gamma_1^0, \ldots, \Gamma_{m_0}^0$, and refine it until we obtain a fixpoint $\Gamma_1^k, \ldots, \Gamma_{m_k}^k$. We decide to start with process p if $|\Delta_{p,q}|_p| < |\Delta_{p,q}|_q|$, i.e., the number of elements paired up in $\Delta_{p,q}$ is smaller for p than for q. Otherwise, we symmetrically start with q. Let the elements of $\Delta_{p,q}|_p$ be $x_1, \ldots x_{m_0}$, and Γ_i^0 be the pairs in $\Delta_{p,q}$ containing x_i. Now, we repeatedly alternate between the q side and the p side the following step: we check for each two sets Γ_i^l and Γ_j^l whether $\Gamma_i^l|_q = \Gamma_j^l|_q$. If it is the case, we combine them into a single set $\Gamma_i^l \cup \Gamma_j^l$. On even steps, we replace q with p. This is done as long as we can unify new subsets in this way. The whole process is performed in time cubic in the size of $\Delta_{p,q}$.

Figure 1 shows the result for an example. The left-hand side represents the coordinators induced by $\Delta_{p,q}$ and the right-side the minimal set of coordinators. Each Γ_i contains a single state of q. And indeed, if we start the procedure with q, the initial partition is already the solution.

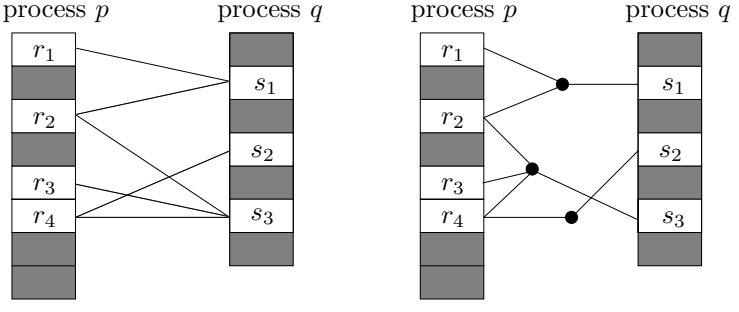


Fig. 1. Minimizing the number of coordinators

4 Knowledge Based Controllers as a Practical Solution for the Distributed Control Problem

We now show some connections between the classical controller synthesis problem (see, e.g., [7]) and knowledge based control. We have provided a solution to the synthesis of distributed controllers, based on adding interactions between transitions in order to combine the knowledge of individual processes. In this section, we want to put the knowledge based solution in the context of the distributed control problem when adding interactions is not allowed. We first show an example where the local knowledge is not sufficient for controlling the system, but where blocking transitions — even when they are known to be maximal — would allow controlling the system. This example shows that distributed controllers are more general than knowledge based controllers. However, there is no algorithm that guarantees constructing them: we show that even our limited problem (and running example) of controlling a system according to priorities is

already undecidable. This advocates that the construction of knowledge based controllers, and furthermore, the use of additional synchronization, is a practical solution for the distributed control problem.

The knowledge approach to control in [6] requires that there is sufficient knowledge to allow *any* transition of the controlled system that does not violate the enforced property ψ. In [1], which we extend here, this requirement is relaxed; the knowledge must suffice to execute *at least one* enabled transition not violating ψ when such a transition exists. In the more general case of distributed controller design, one may want to block some enabled transitions even if their execution does not immediately violate the enforced property. This is required to prevent the transformed system from later reaching deadlocked states, where the controlled system originally had a way to progress (thus, introducing new deadlocks).

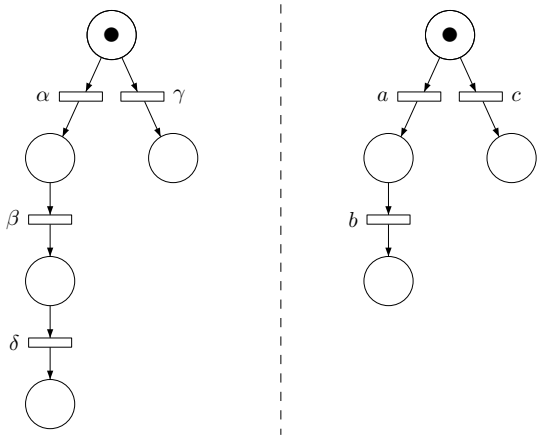


Fig. 2. A system with priorities $\delta \ll b \ll \beta$

Consider a concurrent system, as in Figure 2, with two processes π_l (left) and π_r (right), each one of them having initially a nondeterministic choice. The priorities in this system are $\delta \ll b \ll \beta$. Each process can observe only its own transitions. In the initial state, all four enabled transitions α, γ, a, c are unordered by priorities, and thus are all maximal. If α is fired and subsequently a (or vice versa), we reach a global state where process π_r does not have any enabled transition with maximal priority since $b \ll \beta$. Process π_l does, and it can execute β. Thereafter, since $\delta \ll b$, process π_l cannot execute δ and must wait for process π_r to execute b. Now, with its limited observability, π_l cannot distinguish between the situation before or after b was executed by π_r. Thus π_l lacks the capability, and the corresponding knowledge, of deciding whether to execute δ. In this state, π_r cannot distinguish between the situation before and after β was executed, and cannot decide to execute b. Accordingly, the local knowledge of the processes in this example is not sufficient to construct a controller. In the initial state, both processes can progress freely, only to fall into a situation where they do not know locally when they can safely progress.

When a controller is allowed to block transitions even when their execution does not immediately lead to violation of the property to be preserved, the situation can be recovered. In the example above, we may choose either to block α in favor of γ, or to block a in favor of c. Blocking both α and a is not necessary. This example also shows that there is no *unique maximal* solution to the control problem that blocks the *smallest* number of transitions. Note that an alternative solution to blocking α or a can be achieved using a temporary interaction between the processes, as shown earlier in this paper.

It was shown in [10,8] that the problem of synthesizing a distributed controller is, in general, undecidable. We show here that even when restricting the synthesis problem to priority policies, the problem remains undecidable. The proof for that is given below. Notice that when we have the flexibility of allowing additional coordination, as done in this paper, the problem, in the limit, becomes a sequential control problem, which is decidable.

Theorem 1. *Constructing a distributed controller that enforces a priority policy is undecidable.*

Proof. Following [10], the proof is by reduction from the post correspondence problem (PCP). In PCP, there is a finite set of pairs $\{(l_1, r_1), \ldots, (l_n, r_n)\}$, where the components l_i, r_i are words over a common alphabet Σ, and one needs to decide whether one can concatenate separately a *left word* from the left components and a *right word* from the right components according to a mutual nonempty sequence of indexes $i_1 i_2 \ldots i_k$, such that $l_{i_1} l_{i_2} \ldots l_{i_k} = r_{i_1} r_{i_2} \ldots r_{i_k}$.

Let $i \in \{1..n\}$, $\hat{l}_i$ be the word $l_i i$, i.e., the i^{th} left component concatenated with the index i. Similarly, let $\hat{r}_i$ be $r_i i$. We consider two regular languages: $L = (\hat{l}_1 + \hat{l}_2 + \ldots + \hat{l}_k)^+$ and $R = (\hat{r}_1 + \hat{r}_2 + \ldots + \hat{r}_k)^+$. Now suppose a process π_p executes according to the regular expression $l.L.x.a.b + r.R.x.c.d$. The choice of π_p between l and r is uncontrollable. Suppose also that π_p coordinates (through shared transitions) the alphabet letters from Σ with a process π_{q_1}, and the indexes letters from Σ with another process π_{q_2}. After that, π_{q_1} and π_{q_2} are allowed to interact with each other. Specifically, π_{q_2} sends π_{q_1} the sequences of indexes it has observed. Suppose that now π_{q_1} has a nondeterministic choice between two transitions: α or β. The priorities are set as $b \ll \alpha \ll a$ and $d \ll \beta \ll c$. All other pairs of transitions are unordered according to $\ll$. If π_{q_1} selects α and r was executed, or π_{q_1} selects β and l was executed, then there is no problem, as α is unordered with respect to c and d, and also β is unordered with respect to a and b, respectively. Otherwise, there is no way to control the system so that it executes the sequence $a.\alpha.b$ or $c.\beta.d$ allowed by the priorities.

We show by contrapositive that if there is a controller, then the answer to the PCP problem is negative. Suppose the answer to the PCP problem is positive, i.e., some left and right words are identical and with the same indexes. Then process π_{q_1} cannot make a decision: the information that π_{q_1} observed and later received from π_{q_2} is exactly the same in both cases for the mutual left and right word. Thus, π_{q_1} cannot anticipate whether $c.d$ or $a.b$ will happen and cannot make a safe choice between α and β accordingly.

Conversely, if there is no controller, it means that π_{q_1} cannot make a safe choice between α and β. This can only happen if π_{q_1} and π_{q_2} can observe exactly the same visible information for both an l and an r choice by π_p.

This means that deciding the existence of a controller for this system would solve the corresponding PCP problem. It is thus undecidable.

Note that in this proof we do not ensure a finite memory controller, even when one exists. Indeed, a finite controller may not exist. To see this, assume a PCP problem with one word $\{(a, aa)\}$. To check whether we have observed a left or a right word, we may just compare the number of a's that p has observed with the number of indexes that q has observed.

5 Implementation and Experimental Results

We have implemented a prototype for experimenting with this approach. In our tool, we use Petri nets to represent distributed transition systems.

This tool first builds the set of reachable states and the corresponding local knowledge of each process. Then, it checks whether local knowledge is sufficient to ensure correct distributed execution of the system under study. Let $\mathcal{U}$-states be global states in which *all* corresponding local states satisfy $\neg k_1^p$. The existence of a $\mathcal{U}$-state means that Δ is not an invariant without adding some tuples for synchronization. We allow simulating the system while counting the number of synchronizations and $\mathcal{U}$-states encountered during execution as a measurement of the amount of additional synchronization required.

The example that we used in our experiments is a variant of the dining philosophers where philosophers may arbitrarily take first either the fork that is on their left or right. In addition, a philosopher may hand over a fork to one of his neighbors when his second fork is not available and the neighbor is looking for a second fork as well. Such an exchange (labeled ex) is a way to avoid the well-known deadlocks when all philosophers take first the fork on the same side. This example is partially represented by the Petri net of Figure 3.

In our example, places (concerning philosopher β) are defined as follows:

- $fork^i$: the i-th fork is on the table.
- $0fork_\beta$ (resp. $2forks_\beta$): philosopher β has no fork (resp. 2 forks) in his hands.
- $1fork^l_\beta$ (resp. $1fork^r_\beta$): philosopher β holds his left (resp. right) fork.

Transitions (concerning philosopher β) play the following role:

- get^{kl}_β (resp. get^{kr}_β), $k = 1, 2$: philosopher β takes the fork on his left (resp. on his right). This is his k-th fork.
- $eat\text{-}and\text{-}return_\beta$: philosopher β eats and puts both forks back on the table.
- $ex_{\alpha,\beta}$: philosopher α gives his right fork to philosopher β.
- $ex_{\beta,\alpha}$: philosopher β gives his left fork to philosopher α .

Processes correspond to philosophers. The transitions defining a process β have a β in their name, including the four exchange transitions $ex_{\alpha,\beta}$, $ex_{\beta,\alpha}$, $ex_{\beta,\gamma}$ and $ex_{\gamma,\beta}$.

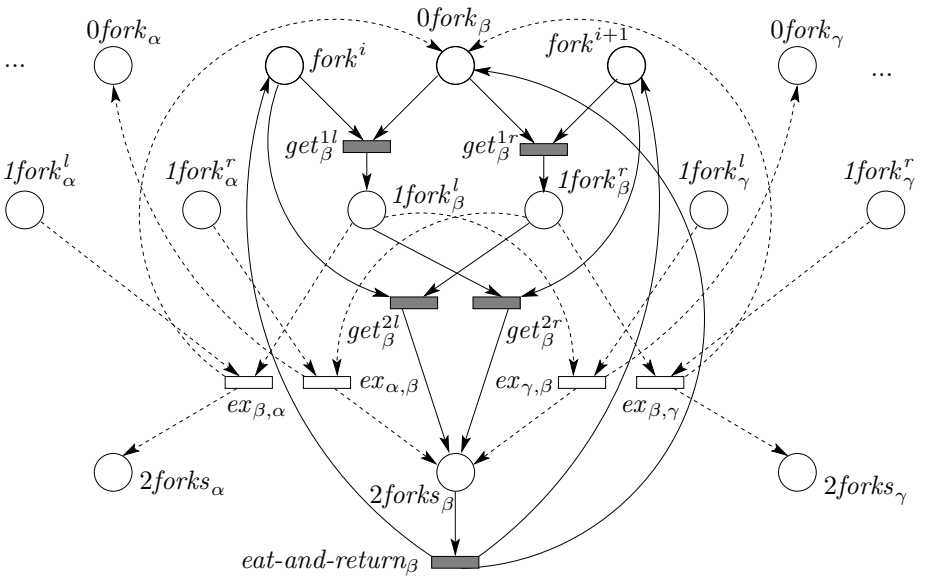


Fig. 3. A partial representation of the dining philosophers (philosopher β)

In Figure 3, transitions related only to philosopher β are drawn with full lines. Transitions in dashed lines are shared between β and one of his neighbors (α on the left, γ on the right).

Not controlling exchanges at all allows nonprogress cycles. To avoid them, we add priorities which allow exchange actions only when a blocking situation has been reached within some degree of locality.

First variant. We use a priority rule stating that an exchange between philosophers α and β has lower priority than α or β taking a fork. This leads to the following priorities for each α and β such that α is βs left neighbor:

- $ex_{\alpha,\beta} \ll get_\alpha^{2l}$: if α can pick up a left fork, he won't give his right fork to β.
- $ex_{\beta,\alpha} \ll get_\beta^{2r}$: symmetrically if β can pick up a right fork.

In this variant, local knowledge is sufficient. Indeed, when a philosopher α and both his neighbors are blocked in a state where they all have a left (resp. a right) fork, then philosopher α has enough knowledge to support an exchange with his left (resp. right) neighbor. For any number of philosophers, there is no $\mathcal{U}$-state. Thus, no extra synchronization is needed.

Second variant. Now, to further reduce the number of exchanges, one may decide that philosopher β may give his left fork to his left neighbor α only if (1) α is blocked (2) β is blocked and (3) βs right neighbor γ is also blocked (the exchange of right forks is similar). This translates into adding the following priorities:

- $ex_{\alpha,\beta} \ll get_\delta^{2l}$, $eat\text{-}and\text{-}return_\delta$ (with δ the left neighbor of philosopher α)
- $ex_{\beta,\alpha} \ll get_\gamma^{2r}$, $eat\text{-}and\text{-}return_\gamma$ (with γ the right neighbor of philosopher β)

Local knowledge alone cannot ensure here correct distributed execution. However, binary synchronizations are sufficient in this example to ensure that the system is always able to move on, for any number of philosophers.

In Table 1, we show results for the second variant with 6, 8 and 10 philosophers. There are two $\mathcal{U}$-states which correspond to the situation where all philosophers hold their left fork, or they all hold their right fork. For computing the number of synchronizations, we used each time 100 runs of a length of 10,000 steps (i.e. transitions). Note that the number of exchange transitions is identical to the number of synchronizations.

Table 1. Results for 100 executions of 10,000 steps for the second variant

philosophers	6	8	10
reachable states	729	6561	59049
synchronizations	354	285	237
$\mathcal{U}$-states encountered	253	149	100

At the current stage, the minimization of the set of coordinators has not been implemented (we use one coordinator per synchronization pair in Δ) and our tool handles only joint local states consisting of two states.

6 Conclusion

Imposing a global constraint upon a distributed system by blocking transitions is, in general, undecidable [10,8]. One practical approach for this problem was to use model checking of knowledge properties [1]. If we allow additional synchronization, the problem becomes decidable: at the limit, everything becomes synchronized, although this, of course, is highly undesirable. The method presented in [1] provided a (disjunctive) controller. The problem with that approach is that in many cases the local knowledge of the separate processes does not suffice. A suggested remedy was to monitor several processes together, achieving this way an increased level of knowledge.

In the current work we look at the situation where we are allowed to coordinate between several processes, but only temporarily. First, we can calculate whether the constraint we want to impose is feasible, when all processes are combined together. This is done using game theory [9]. If this is the case, we check if we can control the system based on the local knowledge of processes or temporary interactions between processes. Of course, our goal is to minimize the number of interactions, and moreover, the number of processes involved in each interaction.

For achieving a distributed implementation, one can use a multiparty synchronization algorithm such as the α-core algorithm [5]. Based on that, we presented an algorithm that uses model checking to calculate when synchronization between local states is needed. The synchronizing processes, successfully coordinating, are then able to use the knowledge table calculated by model checking,

which dictates to them which transition can be executed. Some small corrections to the original presentation of the α-core algorithm appear in [3].

The framework suggested in this paper can be used as a distributed implementation for the Verimag BIP system [2]. BIP is based on a clear separation between the behavior of atomic components and the interaction between such components, which is represented using (potentially hierarchical) connectors. Priorities offer a mechanism to enforce scheduling policies by filtering the set of interactions that can be fired. So far, implementing BIP systems in a distributed setting remains a challenging task.

References

1. Basu, A., Bensalem, S., Peled, D., Sifakis, J.: Priority scheduling of distributed systems based on model checking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 79–93. Springer, Heidelberg (2009)
2. Basu, A., Bozga, M., Sifakis, J.: Modeling heterogeneous real-time components in BIP. In: SEFM, pp. 3–12. IEEE Computer Society Press, Los Alamitos (2006)
3. Katz, G., Peled, D.: Code mutation in verification and automatic code generation. In: TACAS. LNCS. Springer, Heidelberg (to appear, 2010)
4. Orlin, J.B.: Contentment in graph theory: covering graphs with cliques (1977)
5. Pérez, J.A., Corchuelo, R., Toro, M.: An order-based algorithm for implementing multiparty synchronization. Concurrency - Practice and Experience 16(12), 1173–1206 (2004)
6. Rudie, K., Ricker, S.L.: Know means no: Incorporating knowledge into discrete-event control systems. Transactions on Automatic Control 45(9), 1656–1668 (2000)
7. Rudie, K., Wonham, W.M.: Think globally, act locally: decentralized supervisory control. Transactions on Automatic Control 37(11), 1692–1708 (1992)
8. Thistle, J.G.: Undecidability in decentralized supervision. System and Control Letters 54, 503–509 (2005)
9. Thomas, W.: On the synthesis of strategies in infinite games. In: Mayr, E.W., Puech, C. (eds.) STACS 1995. LNCS, vol. 900, pp. 1–13. Springer, Heidelberg (1995)
10. Tripakis, S.: Undecidable problems of decentralized observation and control on regular languages. Inf. Process. Lett. 90(1), 21–28 (2004)
11. van der Meyden, R.: Common knowledge and update in finite environments. Inf. Comput. 140(2), 115–157 (1998)
12. Yoo, T.-S., Lafortune, S.: A general architecture for decentralized supervisory control of discrete-event systems. Discrete Event Dynamic Systems 12(3), 335–377 (2002)

Robustness in the Presence of Liveness*

Roderick Bloem[1], Krishnendu Chatterjee[2], Karin Greimel[1],
Thomas A. Henzinger[2], and Barbara Jobstmann[3]

[1] Graz University of Technology
[2] IST Austria (Institute of Science and Technology Austria)
[3] CNRS/Verimag

Abstract. Systems ought to behave reasonably even in circumstances that are not anticipated in their specifications. We propose a definition of robustness for liveness specifications which prescribes, for any number of environment assumptions that are violated, a minimal number of system guarantees that must still be fulfilled. This notion of robustness can be formulated and realized using a Generalized Reactivity formula. We present an algorithm for synthesizing robust systems from such formulas. For the important special case of Generalized Reactivity formulas of rank 1, our algorithm improves the complexity of [PPS06] for large specifications with a small number of assumptions and guarantees.

1 Introduction

Current verification and synthesis approaches consider the functional correctness of a system as a Boolean question: either the specification is fulfilled, or it is not. This approach is unsatisfactory in many situations [BCHJ09]. In particular, many specifications consist of environment assumptions and system guarantees. For such specifications, the classical approach does not impose any restrictions on the behavior of the system when the environment assumptions are *not* fulfilled. We argue that (1) desirable systems act in some "reasonable" way, even if the environment does not always fulfill the assumptions and (2) it is an undue burden on the user to specify the proper behavior of the system for each and every environment behavior. Desirable systems should fulfill a natural "graceful degradation" property in the sense that the system should fulfill the guarantees as well as it can, given any behavior of the environment.

We have previously studied the verification and synthesis of robust systems for safety specifications [BGHJ09]. In the case of safety, environment failures are immediately apparent and the difficulty is how the system can best recover from them. A violation of a liveness property, however, cannot be detected at any point in time [AS85]. Thus, a system that is robust to liveness failures must attempt to fulfill its guarantees under all circumstances, without knowing whether the environment satisfies the assumptions.

* This work was supported by EU grants 217069 (COCONUT), 248613 (DIAMOND), 215543 (COMBEST), and the European Network of Excellence ArtistDesign.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 410–424, 2010.

In this paper, we define several possible notions of robustness in the presence of liveness, all aiming at maximizing the set of guarantees that is fulfilled for any set of fulfilled assumptions. Suppose a specification has two assumptions and two guarantees. In order for the specification to hold, both guarantees must be met when both assumptions are. A system that meets both guarantees when only one assumption is met is more robust than one that meets one (or zero) guarantees when one assumption is met.

Example 1. We consider a variant of the dining philosophers problem [Dij68]. There are n philosophers sitting at a round table. There is one chopstick between each pair of adjacent philosophers. Because each philosopher needs two chopsticks to eat, adjacent philosophers cannot eat simultaneously. We are interested in schedulers that use input variables h_i signifying that philosopher i is hungry and output variables e_i signifying that philosopher i is eating.

We have the following requirements. First, an eating philosopher prevents her neighbors from eating. Formally, $G_{1i} = \square(e_i \rightarrow \neg e_{(i-1) \bmod n} \wedge \neg e_{(i+1) \bmod n})$. Second, an eating philosopher eats until she is no longer hungry: $G_{2i} = \square(e_i \wedge h_i \rightarrow \bigcirc e_i)$. Third, every hungry philosopher eats eventually $G_{3i} = \square(h_i \rightarrow \diamondsuit e_i)$. We add the assumption that an eating philosopher eventually loses her appetite: $A_{1i} = \square(e_i \rightarrow \diamondsuit \neg h_i)$. Our final specification consists of n assumptions and $3n$ guarantees: $\bigwedge_{i=1}^{n} A_{1i} \rightarrow \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i} \wedge G_{3i})$.

We have synthesized a system realizing this specification for 5 philosophers using our synthesis tool RATSY[1]. The system constructed by RATSY is not very robust: When philosopher 1 violates the assumption by always being hungry, then philosophers 1 and 3 eat forever, while the other philosophers starve. Thus the three guarantees $\square(h_2 \rightarrow \diamondsuit e_2)$, $\square(h_4 \rightarrow \diamondsuit e_4)$, and $\square(h_5 \rightarrow \diamondsuit e_5)$ are violated. A more robust system would let philosopher 3 and 4 take turns, thus violating only two guarantees. □

In this paper, we consider Generalized Reactivity specifications of rank 1 (GR(1) specifications). GR(1) is an expressive specification formalism with a natural distinction between assumptions and guarantees [PPS06]. Efficient tools exist for GR(1) specifications, which have been used to synthesize relatively large specifications [JGWB07, BGJ+07]. GR(1) specifications are of the form $\varphi \rightarrow \psi$. Here, φ represents the environment assumptions and ψ represents the system guarantees and both φ and ψ are given as a set of deterministic Büchi automata. These automata are combined into a product automaton with state space Q, transition relation δ, and acceptance condition $\bigwedge_{i=1}^{m} \square \diamondsuit a_i \rightarrow \bigwedge_{i=1}^{n} \square \diamondsuit g_i$.

GR(1) specifications do not require any guarantees to be fulfilled when some assumption is violated. We propose an intuitive notion of robustness that prescribes, for any number of environment assumptions that is violated, a minimal number of system guarantees that must still be fulfilled. We show that this and related measures of robustness can be transformed to a specification of the form $\bigwedge_{j=1}^{k}(\bigwedge_{i=1}^{m} \square \diamondsuit a_{ji} \rightarrow \bigwedge_{i=1}^{n} \square \diamondsuit g_{ji})$, which is a Generalized Reactivity (generalized Streett) formula of rank k. We address the problem of verification

[1] http://rat.fbk.eu/ratsy/index.php/Main/HomePage

and especially of synthesis of such formulas, which allows us to construct robust systems.

The verification problem is a relatively straightforward generalization of the verification problem for GR(1) (cf. [GBJV08]) and can be performed in time $O(m \cdot n \cdot |Q| \cdot |\delta|)$. Recall that m is the number of assumptions and n is the number of guarantees, and $|Q|$ and $|\delta|$ refer respectively to the size of the state space and the transition relation of the product automaton.

The synthesis question is answered by solving a Generalized Reactivity game. This can either be done through a specialization of Zielonka's algorithm, or through a novel algorithm presented in this paper, both of which can be implemented symbolically. Zielonka's algorithm runs in time $O(|Q|^{2 \cdot k} \cdot |\delta| \cdot (m+n)^k \cdot k!)$, which we improve to $O(|Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$. On the other hand, our algorithm produces larger strategies and thus larger robust systems: the systems produced by Zielonka's algorithm have size $|Q| \cdot n^k \cdot k!$, whereas our algorithm produces systems of size $|Q| \cdot ((m+1) \cdot (n+1))^k \cdot k!$.

Our algorithm is a generalization of a game-theoretic algorithm for the important class of GR(1) conditions based on a reduction (via a counting construction) to Streett games with single pair. The algorithm runs in time $O(|Q| \cdot |\delta| \cdot (m \cdot n)^2)$. This bound improves the $O(|Q|^2 \cdot |\delta| \cdot m \cdot n)$ time bound of the algorithm of [PPS06] for the case that Q is larger than m and n, which is typical in such applications as GR(1) synthesis.

Measures of robustness for different fault models, for example internal malfunctions of circuits [FD08], have been studied. Classical notions of fault tolerance such as self-stabilization [Dij74] and the notions of closure and convergence suggested in [Aro93] focus on safety properties. Convergence requires that a system restores its invariant after an error has occurred, and closure requires that the system satisfies a second, larger invariant even when errors recur. Our approach can be viewed as an extension of closure to liveness, where we require that some weaker set of guarantees is fulfilled when the environment behaves unexpectedly. Apart from our previous work [BGHJ09], there is little work on synthesis of robust systems, although people have studied the related problem of retrofitting fault tolerance to existing programs. (See, e.g., [KE05, EKA08].)

The flow of the paper is as follows. After giving the necessary notation in Section 2, we define several notions of robustness in Section 3. In order to solve the synthesis problem for robust systems, we introduce the necessary transformations on the formulas and game theoretic algorithms in Sections 4 and 5. In Section 6 we return to the questions of verification and synthesis of robust systems. We conclude with Section 7.

2 Preliminaries

We consider systems with a set of input signals I and a set of output signals O. We define $AP = I \cup O$. We use the signals as atomic propositions in the specifications defined below. Our input alphabet is thus $\Sigma_I = 2^I$, the output alphabet is $\Sigma_O = 2^O$, and we define $\Sigma = 2^{AP}$.

Acceptance Conditions. The specifications we use are automata and we synthesize a system that realizes a given specification using games. Both automata and games can have the following acceptance conditions. Let Q be a set of states, an *acceptance condition* is a predicate $\mathrm{Acc} : Q^\omega \to \mathbb{B}$, mapping infinite runs to true or false (accepting and not accepting, or winning and losing, respectively). The *Büchi acceptance condition* is $\mathrm{Acc}(\rho) = 1$ iff $\inf(\rho) \cap F \neq \emptyset$, where $F \subseteq Q$ is the set of accepting states and $\inf(\rho)$ is the set of elements that occur infinitely often in ρ. We abbreviate the Büchi condition as $\mathcal{B}(F)$. A *Generalized Reactivity acceptance condition* is a predicate $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m_l} \mathcal{B}(A_{l,i}) \to \bigwedge_{i=1}^{n_l} \mathcal{B}(G_{l,i}))$, where $A_{l,i} \subseteq Q$ are assumptions and $G_{l,i} \subseteq Q$ are guarantees. To simplify notation, we will assume that the m_l are all equal to some constant m, and similarly for n_l and n. The acceptance condition is a *GR(1) acceptance condition* if $k = 1$, it is a *generalized Büchi acceptance condition* if $k = 1$ and $m = 0$, it is a *Streett acceptance condition* with k pairs if $m = n = 1$.

Automata. A *(complete deterministic) automaton* A over the alphabet Σ is a tuple $A = (Q, q_0, \delta, \mathrm{Acc})$, where Q is a finite set of states, $q_0 \in Q$ is the initial state, $\delta : Q \times \Sigma \to Q$ is the transition function, and Acc is the acceptance condition. A *run* of an automaton A on a word $w = w_0 w_1 \ldots \in \Sigma^\omega$ is the sequence $\rho(w) = \rho_0 \rho_1 \ldots \in Q^\omega$ such that $\rho_0 = q_0$, and $\rho_{i+1} = \delta(\rho_i, w_i)$. An automaton accepts a word if its run is accepting ($\mathrm{Acc}(\rho(w)) = 1$); its language $L(A)$ consists of the set of words it accepts. A *Büchi automaton* $A = (Q, q_0, \delta, F)$ is an automaton with a Büchi condition with accepting state set F.

Generalized Reactivity(1) specifications. Consist of two parts: *assumptions* and *guarantees* [PPS06]. They specify the interaction between an environment (controlling the input variables Σ_I) and a system (controlling the output variables Σ_O). The specification states that the system must fulfill all guarantees whenever the environment fulfills all assumptions.

A GR(1) specification over the alphabet Σ consists of m Büchi automata $A_1^a, \ldots, A_m^a$ for the environment assumptions and n Büchi automata $A_1^g, \ldots, A_n^g$ for the system guarantees. [PPS06]. Let $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$ be the product of all automata A_i^a and A_i^g, where the state space is $Q = Q_1^a \times \cdots \times Q_m^a \times Q_1^g \times \cdots \times Q_n^g$, the transition function is $\delta((q_1^a, \ldots, q_n^g), \sigma) = (\delta_1^a(q_1^a, \sigma), \ldots, \delta_n^g(q_n^g, \sigma))$, and the initial state is $q_0 = (q_{0,1}^a, \ldots, q_{0,n}^g)$. Let $J_i^a = \{(q_1^a, \ldots, q_n^g) \in Q \mid q_i^a \in F_i^a\}$ be the set of states that are accepting in A_i^a. Similarly, let J_i^g be the set of all states of $A^{GR(1)}$ that are accepting in A_i^g. The acceptance condition Acc is a GR(1) condition with assumptions J_i^a and guarantees J_i^g.

Note that the size of the state space of the specification grows exponentially with the number of assumptions and guarantees (if the Büchi automata have more than 2 states), whereas m and n grow linearly.

A system realizes a GR(1) specification $A^{GR(1)}$ if the language of the system is part of the language of $A^{GR(1)}$.

Games and Strategies. A *game graph* is a finite directed graph $G = (S, s_0, E)$ consisting of a set of states S, an initial state $s_0 \in S$, and a set of edges $E \subseteq S \times S$ such that each state has at least one outgoing edge. The states are partitioned

into a set S_1 of *Player-1 states* and a set S_2 of *Player-2 states*. When the initial state is not relevant, we omit it and write (S, E). A *play* $\rho = s_0 s_1 \ldots \in S^\omega$ is an infinite sequence of states such that for all $i \geq 0$ we have $(s_i, s_{i+1}) \in E$. Given a game graph $G = (S, E)$, a *(finite memory) strategy* for Player 1 is a tuple (Γ, γ_0, π), where Γ is some (finite) set representing the memory, $\gamma_0 \in \Gamma$ is the initial memory content, and $\pi : S_1 \times \Gamma \to S \times \Gamma$ is a function mapping a Player-1 state s and a memory content to a successor state s' and an updated memory content such that $(s, s') \in E$. A Player-2 strategy is defined similarly. A strategy is *positional* if it depends only on the current state. We represent a positional strategy π for player p as a function from S_p to S. Let $\rho((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s)$ denote the unique play starting at s when Player 1 plays according to the strategy $(\Gamma_1, \gamma_{0,1}, \pi_1)$ and Player 2 plays according to $(\Gamma_2, \gamma_{0,2}, \pi_2)$.

A *game* is a tuple $((S, E), \mathrm{Acc})$, consisting of a game graph (S, E) and an objective Acc. The game graph defines the possible actions of the players. The objective describes the winning condition for the players. A play ρ is winning for Player 1 if it satisfies the objective of the game, otherwise it is winning for Player 2. A strategy π_1 is winning for Player 1 if for all strategies π_2 of Player 2 the play $\rho((\Gamma_1, \gamma_{0,1}, \pi_1), (\Gamma_2, \gamma_{0,2}, \pi_2), s_0)$ is winning. A game is winning for Player 1 (Player 2) if there exists a winning strategy for Player 1 (Player 2, resp.). A *Generalized Reactivity (GR) game* is a game with a Generalized Reactivity acceptance condition, and similarly for GR(1).

Given a game graph G, two objectives are equivalent if all plays in G have the same winner for both objectives. The objectives are *equivalent* if they are equivalent for any game graph.

GR(1) Synthesis. A GR(1) specification can easily be translated into a GR(1) game. A winning strategy for the GR(1) game corresponds to a system that realizes the GR(1) specification.

3 Defining Measures of Robustness

In this section we discuss how to compare systems with respect to robustness. Usually, multiple systems satisfy a specification, but which one is most robust? In prior work we answered this question for safety specifications: our measure of robustness for a safety specification $\varphi \to \psi$ is the ratio between how often the environment violates φ and how often the system violates ψ. For specifications with liveness properties, this approach does not work because we cannot count the number of violations of a liveness property. Instead, we propose to count the number of properties violated. In the following we show two different robustness measures, the single and the multiple counting requirements measure. Then we formally state the requirements a robustness measure has to satisfy.

Single Counting Requirements. Recall the dining philosophers example with $n = 5$ philosophers given in the introduction. Suppose system D_1 always lets one philosopher eat until she is not hungry anymore and then moves to the next hungry philosopher in a round robin manner. If one philosopher is hungry

forever, then no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of four guarantees.

Suppose system D_2 lets two non-adjacent philosophers eat at the same time until neither is hungry anymore. They take turns in the following order: first philosopher 1 and 3 eat, then philosopher 2 and 4, and last philosopher 3 and 5 eat. If one of the currently eating philosopher is hungry forever, then the two currently eating philosophers eat forever and no other philosopher gets to eat again. Thus, the violation of one assumption leads to the violation of three guarantees. System D_2 is thus more robust than system D_1.

An even more robust system (D_3) is the one described in the introduction. Two philosophers eat at the same time, as soon as one of them is not hungry anymore another philosopher with free chopsticks is allowed to eat. If one philosopher is hungry forever, she eats forever and the other philosophers that are not her neighbors take turns eating. The violation of one assumption leads to the violation of two guarantees.

We specify robust systems by adding restrictions to the original specification. All three systems above satisfy the original specification $\varphi = \bigwedge_{i=1}^{n} A_{1i} \rightarrow \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i} \wedge G_{3i})$, but only D_2 and D_3 guarantee that they violate at most three system guarantees if the environment violates one of its assumptions. Formally, D_2 and D_3 additionally satisfy

$$\psi_1 = \Big(\bigvee_{i=1}^{n} \bigwedge_{j\in\{1,\ldots,n\}\setminus\{i\}} A_{1j}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n}\bigvee_{k=j+1}^{n} \bigwedge_{l\in\{1,\ldots,n\}\setminus\{i,j,k\}} G_{3l}\Big),$$

where $\varphi_S = \bigwedge_{i=1}^{n}(G_{1i} \wedge G_{2i})$. The antecedent of the formula states that the environment satisfies $n-1$ out of the n assumptions. The consequent says that the system satisfies all the safety guarantees (G_{1i} and G_{2i}) but might violate three of its liveness guarantees.

Note that in general, a robust system cannot violate a safety guarantee in response to a violation of a fairness assumption, since a violation of a fairness assumption can not be detected in finite time.

Since D_3 violates at most two system guarantees if one environment assumption is violated, it also satisfies the following formula.

$$\psi_2 = \Big(\bigvee_{i=1}^{n} \bigwedge_{j\in\{1,\ldots,n\}\setminus\{i\}} A_{1j}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n} \bigwedge_{k\in\{1,\ldots,n\}\setminus\{i,j\}} G_{3k}\Big)$$

These two formulas allow us to distinguish between systems D_1, D_2, and D_3, which satisfy the same base specification but differ in how resilient they are with respect to violated environment assumptions. We propose to use formulas of this type, which relate the number of satisfied assumptions to a number of satisfied guarantees to measure how robust a system is.

Suppose $\mathcal{A}$ is a set of assumptions and $\mathcal{G}$ is a set of guarantees. Let $\mathcal{A}_k = \{A \subseteq \mathcal{A} \mid |A| = k\}$ be the set of all subsets of $\mathcal{A}$ of size k and let $\mathcal{G}_k$ be defined similarly. We can augment the specification with a restriction of the form $(\bigvee_{A\in\mathcal{A}_k} \bigwedge_{A_i\in A} A_i) \rightarrow (\bigvee_{G\in G_l} \bigwedge_{G_i\in G} G_i)$ to check if a system satisfies l

guarantees when k assumptions are satisfied. Naturally, a system that satisfies more guarantees with the same number of satisfied assumptions is more robust.

Multiple Counting Requirements. In some cases we might want to have a more fine-grained measure of robustness, which cannot be expressed by a single restriction of the form given above. Recall again the dining philosophers example but this time assume there are $n = 7$ philosophers. Suppose system D_4 allows two hungry philosophers to eat at the same time. Then, even if one philosopher does not stop eating, the other non-adjacent philosophers can still take turns eating. However, if two philosophers misbehave and they both get to eat (i.e., they do not sit next to each other), they will leave the other five philosophers to starve. Suppose another system D_5 allows three philosophers to eat at the same time. Now, if two philosophers misbehave and they both get to eat, the system D_5 still allows another philosopher to eat and only four philosophers are left to starve. Both D_4 and D_5 realize the specification φ. If we consider the restrictions from above, we see that both systems satisfy the formula ψ_1 and ψ_2. Our previous measure of robustness cannot distinguish between D_4 and D_5. Let's add another restriction ψ_3 to our specification:

$$\psi_3 = \Big(\bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n}\bigwedge_{k\in\{1,\dots,n\}\setminus\{i,j\}} A_{1k}\Big) \rightarrow \Big(\varphi_S \wedge \bigvee_{i=1}^{n}\bigvee_{j=i+1}^{n}\bigvee_{k=j+1}^{n}\bigwedge_{l\in\{1,\dots,n\}\setminus\{i,j,k\}} G_{3l}\Big)$$

System D_5 realizes $\varphi\wedge\psi_2\wedge\psi_3$ but system D_4 does not. We can measure the number of satisfied guarantees for several numbers of satisfied assumptions. The restrictions we add to the specifications are of the form $\bigwedge_{(k,l)\in L}((\bigvee_{\mathcal{A}\in\mathcal{A}_k}\bigwedge_{A_i\in\mathcal{A}} A_i) \rightarrow (\bigvee_{\mathcal{G}\in\mathcal{G}_l}\bigwedge_{G_i\in\mathcal{G}} G_i))$, where L is a list of pairs (k,l), requiring l guarantees to be satisfied if k assumptions are satisfied.

Definitions. Both single and multiple counting requirements, as defined above, can be put in the following form (as we will shown in Section 4).

Definition 1. *Given a GR(1) specification $A^{GR(1)}$ with assumptions $J^a_1,\dots,J^a_m$ and guarantees $J^g_1,\dots,J^g_n$, a* robustness specification *for $A^{GR(1)}$ has the form*

$$\bigwedge_{l=1}^{k}\left(\bigwedge_{i=1}^{m_l}\mathcal{B}(J^a_{l,i}) \rightarrow \bigwedge_{i=1}^{n_l}\mathcal{B}(J^g_{l,i})\right),$$

where $J^a_{l,i}\in\{J^a_1,\dots,J^a_m\}$ and $J^g_{l,i}\in\{J^g_1,\dots,J^g_n\}$.

There is a natural partial order on robustness specifications: If, for each set of satisfied assumptions, a specification S requires a superset of the guarantees required by specification S', then S is more robust than S'. Let us denote this order by $\prec$.

Definition 2. *A* robustness measure *for a GR(1) specification is a set of robustness specifications together with a total order that respects $\prec$.*

For example, consider again the 'simple counting requirements' robustness specifications from above. A possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = 0, l = 1) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}|, l = 0)$, where k is the number of satisfied assumptions and l the number of satisfied guarantees. Another possible total order is $(k = 0, l = |\mathcal{G}|) > (k = 1, l = |\mathcal{G}|) > \ldots > (k = |\mathcal{A}| - 1, l = |\mathcal{G}|) > (k = 0, l = |\mathcal{G}| - 1) > \ldots > (k = |\mathcal{A}|, l = 0)$. A total order is necessary to synthesize the most robust specification.

Section 6 shows how to verify and synthesize robust systems for a given measure. To synthesize a robust system, we solve games with the robustness specification as objective. Section 5 shows how to solve such games. In the next section, we show how to translate combinations of Büchi objectives to generalized Büchi objectives.

4 Simplification of Combinations of Büchi Objectives

In this section we present a simplification of disjunctions of conjunctions of Büchi objectives (DCB objectives) to conjunctions of Büchi objectives (generalized Büchi objectives). This simplification is needed to transform counting requirements to robustness specifications. The simplification (or reduction) incurs an exponential blowup. Games with generalized Büchi objectives can be solved in polynomial time, whereas we show that games with DCB objectives are coNP-complete. This shows that the exponential blow up in the translation is probably inevitable.

Simplification of DCB objectives. The simplification is done in two steps. First, we show how to translate DCB objectives to conjunctions of disjunctions of Büchi objectives. Second, we show that conjunctions of disjunctions of Büchi objectives can be translated to generalized Büchi objectives.

Lemma 1. *Any winning condition ψ that is a DCB objective can be translated into an equivalent winning condition ψ' that is a conjunction of disjunctions of Büchi objectives, such that $|\psi'| = O(2^{|\psi|})$.*

Proof. For any objective $\psi = \bigvee_{i=1}^{m} \bigwedge_{j=1}^{n} \mathcal{B}(B_{ij})$ there exists an equivalent objective $\psi' = \bigwedge_{i=1}^{n^m} \bigvee_{j=1}^{m} \mathcal{B}(B'_{ij})$ with $B'_{ij} \in \{B_{ij} \mid i \in \{1 \ldots m\} \text{ and } j \in \{1 \ldots n\}\}$. The translation is identical to that of changing DNF into CNF. □

Lemma 2. *Any winning condition ψ that is a conjunction of disjunctions of Büchi objectives can be translated into an equivalent generalized Büchi objective ψ', such that $|\psi'| = O(|\psi|)$.*

Proof. Since a disjunction of Büchi conditions is again a Büchi condition $(\mathcal{B}(B_1) \vee \mathcal{B}(B_2) = \mathcal{B}(B_1 \cup B_2))$, objectives of the form $\bigwedge_{i=1}^{k} \bigvee_{j=1}^{l} \mathcal{B}(B_{ij})$ can be reduced to a generalized Büchi objective $\bigwedge_{i=1}^{k} \mathcal{B}(\bigcup_{j=1}^{l} B_{ij})$. □

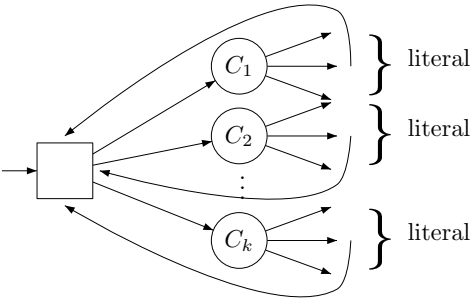


Fig. 1. Game graph for 3SAT formula

Corollary 1. *Any winning condition ψ that is a DCB objective can be translated into an equivalent generalized Büchi objective ψ', such that $|\psi'| = O(2^{|\psi|})$.*

Complexity of solving DCB objectives. We first show that the problem of deciding whether Player 1 has a winning strategy for a DCB objective is coNP-hard, and then we will argue coNP-completeness.

Hardness proof. We show that the problem of deciding whether Player 1 has a winning strategy in a game with a DCB objective is at least as hard as deciding whether a 3SAT formula is unsatisfiable. Consider a 3SAT formula ψ in CNF with clauses $C_1, C_2, \ldots, C_k$ over variables $\{x_1, x_2, \ldots, x_n\}$, where each clause consists of disjunctions of exactly three literals (a literal is a variable or its complement). Given the formula ψ, we construct a game graph as shown in Figure 1. The game graph is as follows: from the initial state, Player 1 chooses a clause, then from a clause Player 2 chooses a literal that appears in the clause (i.e., makes the clause true). From every literal the next state is the initial state. The winning condition for Player 1 is $\bigvee_{i=1}^{n}(\mathcal{B}(X_i) \wedge \mathcal{B}(\overline{X_i}))$, where X_i is the set of states that correspond to the literal x_i and $\overline{X_i}$ is the set of states that correspond to the complement literal $\neg x_i$; in other words, Player 1 wants to visit some variable and its complement infinitely often.

We now present two directions of the hardness proof.

Not satisfiable implies winning. We show that if ψ is not satisfiable, then Player 1 has a winning strategy. The winning strategy is as follows: the strategy is played in rounds; in round i Player 1 chooses the clauses $C_1, C_2, \ldots, C_k$ in order, and then proceeds to round $i + 1$. Since ψ is not satisfiable, for every round i there is at least one variable such that both the variable state and its complement state is visited in round i. Since the number of variables is finite, it follows that there must be some variable such that both the variable state and its complement state appears infinitely often. The result follows.

Satisfiable implies not winning. We now show that if ψ is satisfiable, then Player 2 has a winning strategy. Consider a satisfying assignment to ψ. A memoryless winning strategy for Player 2 is as follows: for every clause C_i, Player 2 chooses a literal from C_i that is set true by the satisfying assignment. Given the

strategy of Player 2, since the strategy is obtained from a valid assignment, it follows that never a variable and its complement is visited.

The above argument gives us the following lemma.

Lemma 3. *Given a game graph with a DCB objective, deciding if Player 1 has a winning strategy is coNP-hard.*

Lemma 4. *Given a game graph with a DCB objective, deciding if Player 1 has a winning strategy can be achieved in coNP.*

Proof. The proof is as follows: we have already shown that DCB objectives can be translated to a generalized Büchi objective (which is an upward-closed objective). It follows from the result of Zielonka [Zie98] that there are memoryless winning strategies for the complement of an upward-closed objective (in particular for disjunction of coBüchi objectives). It follows that there always exist memoryless winning strategies for Player 2. Hence to falsify that Player 1 has a winning strategy, a memoryless strategy for Player 2 can be fixed (as the polynomial witness) and the resulting one-player graph can be verified in polynomial time. The polynomial time verification procedure uses the following fact: consider a maximal strongly connected component (MSCC) in a one-player graph (only Player 1), then the MSCC is winning if for some index i of the disjunction, for every index j of the corresponding conjunction the MSCC contains at least one Büchi state B_{ij}. Using the above fact, MSCC decomposition of a graph, and reachability to winning MSCCs we obtain a polynomial time verification procedure. The result follows. □

Lemma 3 and Lemma 4 yield the following result.

Theorem 1. *Given a game graph with a DCB objective for Player 1, deciding if Player 1 has winning strategy is co-NP complete.*

5 Solving Generalized Reactivity Games

In this section, we first present a translation of GR(1) winning conditions to one-pair Streett conditions (or parity $\{0, 1, 2\}$ conditions). Our reduction is based on a counting construction similar to the reduction of generalized Büchi conditions to Büchi conditions. Second, we generalize the translation to reduce games with Generalized Reactivity objectives to games with Streett objectives.

Reduction. Consider a GR(1) game $G = ((S, E), \mathrm{Acc})$ with $\mathrm{Acc} = \bigwedge_{i=1}^{m} \mathcal{B}(A_i) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_i)$ with Player 1 states S_1 and Player 2 states S_2. We construct an equivalent one-pair Streett game $G' = ((S', E'), \mathcal{B}(A'_1) \rightarrow \mathcal{B}(G'_1))$ with Player 1 states S'_1 and Player 2 states S'_2 as follows.

1. The state space $S' = S \times \{0, 1, \ldots, m\} \times \{0, 1, \ldots, n\}$, with $S'_1 = S_1 \times \{0, 1, \ldots, m\} \times \{0, 1, \ldots, n\}$, and $S'_2 = S_2 \times \{0, 1, \ldots, m\} \times \{0, 1, \ldots, n\}$.
2. The set of edges E' is defined as follows:

$((s,i,n),(s,0,0)) \in E'$ for $0 \leq i \leq m$,
$((s,m,j),(s,0,j)) \in E'$ if $j \neq n$, and
$((s,i,j),(s',i',j')) \in E'$ if $(s,s') \in E, i' = i+1$ if $s' \in A_{i+1}$ otherwise $i' = i$, and $j' = j+1$ if $s' \in G_{j+1}$ otherwise $j' = j$.

3. The Streett pair is $(A'_1 = \{(s,m,j) \in S' \mid j \in \{0,\ldots,n\}\}, G'_1 = \{(s,i,n) \in S' \mid i \in \{0,\ldots,m\}\})$.

We present the intuition behind the construction. Initially i and j are zero. If all the assumptions are visited such that, assumption A_2 is visited after some visit to assumption A_1; assumption A_3 is visited after some visits to assumptions A_1, A_2; assumption A_4 is visited after some visits to assumptions A_1, A_2, A_3; and so on, since the last reset, then i is reset to 0. If all the guarantees are visited, such that guarantee G_2 is visited after some visit to guarantee G_1; guarantee G_3 is visited after some visits to guarantees G_1, G_2; guarantee G_4 is visited after some visits to guarantees G_1, G_2, G_3; and so on, since the last reset, then j is reset to 0. In between resets, i and j denote the last assumption and the last guarantee visited in the order described above, since the last reset. The size of the new state space is $|S'| = |S| \cdot (m+1) \cdot (n+1) = O(|S| \cdot m \cdot n)$. The new number of transitions is $|E'| = |E| \cdot (m+1) \cdot (n+1) + 2 \cdot |S| = O(|E| \cdot m \cdot n)$.

Lemma 5. *There exists a winning strategy for G iff there exists a winning strategy for G'.*

Proof. Consider a play ρ in G and the corresponding play ρ' in G'. We consider two cases. Case one. We consider the case where all guarantees appear infinitely often in ρ. If all guarantees are visited infinitely often, then a state with third state component with value n is visited infinitely often in ρ' (i.e., G'_1 is visited infinitely often). Thus, if the play in G satisfies the GR(1) condition by visiting all guarantees infinitely often, then the corresponding play in G' visits G'_1 infinitely often and satisfies the Streett condition.

Case two. We consider the case where some guarantee is not visited infinitely often in ρ. In this case a state with third state component with value n is visited only finitely often in ρ'. We consider two sub-cases.

Case two(a). If all the assumptions are visited infinitely often in ρ, then a state with second state component with value m is visited infinitely often in ρ'. In this case the play in G does not satisfy the GR(1) condition, and the corresponding play in G' visits A'_1 infinitely often and G'_1 finitely often, which violates the Streett condition.

Case two(b). If some assumption is not visited infinitely often in ρ, then a state with second state component with value m is visited only finitely often in ρ' (i.e., A'_1 is visited finitely often). In this case the play in G satisfies the GR(1) condition, and the corresponding play in G' satisfies the Streett condition. This completes the proof. □

Theorem 2. *Games with GR(1) objectives can be solved in $O(|S| \cdot |E| \cdot (m \cdot n)^2)$ time.*

Since one-pair Streett (or parity $\{0,1,2\}$)) games with $|S|$ states and $|E|$ edges can be solved in $O(|S| \cdot |E|)$ time [Jur00], from Lemma 5 we obtain the above theorem. It may also be noted that one-pair Streett games can be solved very efficiently in practice [dAF07] and also symbolically [EJ91] (and implementing our counting construction symbolically is standard). The previous best know algorithm to solve GR(1) games was through the triple nested fix-point algorithm of [PPS06] which works in time $O(|S|^2 \cdot |E| \cdot n \cdot m)$. For the typical case that $|S|$ is much greater than m and n, our algorithm is faster.

Our algorithm can easily be generalized to Generalized Reactivity objectives.

Theorem 3. *Games with Generalized Reactivity objectives can be solved in* $O(|S|^k \cdot |E| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ *time.*

Proof. Turn all GR(1) objectives into Streett pairs, the Streett game has $O(|S| \cdot m^k \cdot n^k)$ states, $O(|E| \cdot m^k \cdot n^k)$ transitions, and k-Streett pairs. A Streett game with k pairs, $|E'|$ transitions and $|S'|$ states can be solved in $O(|E'| \cdot |S'|^k \cdot k!)$ [PP06]. □

A symbolic algorithm for Generalized Reactivity objectives can be obtained as follows: use the standard symbolic implementation of the counting construction along with the symbolic algorithm for Streett games from [PP06]. This gives us a symbolic algorithm for solving games with Generalized Reactivity objectives.

Winning strategy and memory required. A winning strategy for a GR(k) condition is obtained as follows: first we consider an automaton A_1 of size $((n+1) \cdot (m+1))^k$ to store the values of the counters and follow the transition as given in the reduction to Streett games with k pairs (essentially this mimics the reduction of the counting construction). Winning strategies in Streett games with k pairs require at most $k!$ memory, and a winning strategy (automata A_2 with $k!$ memory) can be constructed from the Streett game solving algorithms (such as [PP06] or [CHP07]). The product automaton $A_1 \times A_2$ describes a winning strategy for the GR(k) condition and requires $((n+1) \cdot (m+1))^k \cdot k!$ memory.

In the case of GR(1) conditions, our construction of winning strategies requires $(n+1) \cdot (m+1)$ memory. The memory can be improved to n as follows: once the winning set is computed, we can run Zielonka's algorithm to compute a winning strategy with n memory. However, as the winning set is already computed we can get rid of the outer iteration of Zielonka's algorithm and re-running Zielonka's algorithm to compute the winning strategy, given the winning set, takes $O(|S| \cdot |E| \cdot (n+m))$ time.

6 Verification and Synthesis of Robust Systems

First, we show how to verify whether a system has a certain level of robustness. Then, we give an algorithm to synthesize the most robust system with respect to a given robustness measure.

Verification. Verification of a robustness specification is similar to the verification of a GR(1) specification.

Lemma 6. *Given a GR(1) specification* $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$ *with* m *assumptions and* n *guarantees, and a system* M*, verification can be performed in* $O(m \cdot n \cdot |Q|^2 \cdot |\delta|)$ *time.*

Proof. Check if a trace in $A^{GR(1)} \times M$ satisfies $\bigwedge_{i=1}^{m} \mathcal{B}(A_i) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_i))$ (the negation of the specification) using the μ-calculus formula $\mu X.(pre(X) \vee \bigvee_{j=1}^{n} \nu Y.(\neg G_j \wedge \bigwedge_{i=1}^{m} pre(\mu Z.(Y \wedge (A_i \vee pre(Z))))))$ [GBJV08]. The complexity of the nested fix-points is in $O(|Q|^2 \cdot |\delta|)$ [EL86]. □

Theorem 4. *Given a GR(1) specification* $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$*, a robustness specification* $\bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$*, and a system* M*, verifying that* M *satisfies the robustness specification takes* $O(k \cdot m \cdot n \cdot |Q|^2 \cdot |\delta|)$ *time.*

Proof. Check if a trace in $A^{GR(1)} \times M$ satisfies the negation of the specification $\bigvee_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_{l,j})))$ by checking the k different GR(1) parts $(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \wedge (\bigvee_{i=1}^{n} \neg \mathcal{B}(G_{l,j})))$ separately, one after the other, using the method of Lemma 6. □

Synthesis. The most robust system with respect to a given robustness measure can be synthesized by synthesizing the greatest realizable robustness specification. Thus, synthesis can be reduced to solving GR games.

Theorem 5. *Given a GR(1) specification* $A^{GR(1)} = (Q, \delta, q_0, \mathrm{Acc})$*, and a robustness measure with* h *robustness specifications* $r_p = \bigwedge_{l=1}^{k}(\bigwedge_{i=1}^{m} \mathcal{B}(A_{l,i}) \rightarrow \bigwedge_{i=1}^{n} \mathcal{B}(G_{l,i}))$*, with* $1 \leq p \leq h$*, and a total order, synthesis of the most robust system can be performed in* $O(h \cdot |Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ *time. The size of the resulting system is* $((m+1) \cdot (n+1))^k \cdot k! \cdot |Q|$*.*

Proof. The best system can be synthesized by trying the specifications in order. Start with the lagest robustness specification according to the given total order. Try to synthesize a system satisfying the specification using the algorithm given in Section 5. The translation of the specification into a game graph is linear, hence synthesis of a robustness specification can be performed in $O(|Q|^k \cdot |\delta| \cdot (m \cdot n)^{k \cdot (k+1)} \cdot k!)$ time (see Theorem 3). The size of the synthesized system is $((m+1) \cdot (n+1))^k \cdot k! \cdot |Q|$, if the robustness specification is realizable. If the robustness specification is not realizable proceed with the next specification in the given order. □

7 Conclusions

We have presented a framework for robustness for liveness specifications. The notion of robustness that we suggest aims to maximize the number of guarantees that are fulfilled for any number of assumptions that may be violated. We have discussed several different interpretations of this notion and have shown that they can all be reduced to Generalized Reactivity formulas. We have shown how to verify such formulas and how to synthesize them to robust systems.

For synthesis we have developed a novel game-theoretic algorithm that is faster than Zielonka's, although it does produce strategies with larger memory. Our algorithm can also be used for the synthesis of GR(1) properties, in which case it outperforms the algorithm of [PPS06] when the state space of the specification is larger than the number of assumptions and guarantees.

References

[Aro93] Arora, A.: Closure and convergence: A foundation of fault-tolerant computing. IEEE Transatcions of Software Engineering 19, 1015–1027 (1993)

[AS85] Alpern, B., Schneider, F.B.: Defining liveness. Information Processing Letters 21, 181–185 (1985)

[BCHJ09] Bloem, R., Chatterjee, K., Henzinger, T., Jobstmann, B.: Better quality in synthesis through quantitative objectives. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 140–156. Springer, Heidelberg (2009)

[BGHJ09] Bloem, R., Greimel, K., Henzinger, T., Jobstmann, B.: Synthesizing robust systems. In: Proc. Formal Methods in Computer Aided Design (FMCAD), pp. 85–92 (2009)

[BGJ+07] Bloem, R., Galler, S., Jobstmann, B., Piterman, N., Pnueli, A., Weiglhofer, M.: Automatic hardware synthesis from specifications: A case study. In: Proceedings of the Design, Automation and Test in Europe, pp. 1188–1193 (2007)

[CHP07] Chatterjee, K., Henzinger, T.A., Piterman, N.: Generalized parity games. In: Seidl, H. (ed.) FOSSACS 2007. LNCS, vol. 4423, pp. 153–167. Springer, Heidelberg (2007)

[dAF07] de Alfaro, L., Faella, M.: Accelerated algorithms for 3-color parity games with an application to timed games. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 108–120. Springer, Heidelberg (2007)

[Dij68] Dijkstra, E.W.: Cooperating sequential processes. In: Genuys (ed.) Programming Languages, pp. 43–112. Academic Press, London (1968)

[Dij74] Dijkstra, E.: Self-stabilizing systems in spite of distributed control. Communications of the ACM 17, 643–644 (1974)

[EJ91] Emerson, E.A., Jutla, C.S.: Tree automata, mu-calculus and determinacy. In: Proc. 32nd IEEE Symposium on Foundations of Computer Science, October 1991, pp. 368–377 (1991)

[EKA08] Ebnenasir, A., Kulkarni, S.S., Arora, A.: Ftsyn: a framework for automatic synthesis of fault-tolerance. Software Tools for Technology Transfer 10, 455–471 (2008)

[EL86] Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional mu-calculus. In: Proceedings of the First Annual Symposium of Logic in Computer Science, June 1986, pp. 267–278 (1986)

[FD08] Fey, G., Drechsler, R.: A basis for formal robustness checking. In: ISQED, pp. 784–789 (2008)

[GBJV08] 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)

[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)

[Jur00] 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)

[KE05] Kulkarni, S.S., Ebnnenasir, A.: Complexity issues in automated synthesis of failsafe fault-tolerance. IEEE Transactions on Dependable and Secure Computing 2, 1–15 (2005)

[PP06] Piterman, N., Pnueli, A.: Faster solutions of Rabin and Streett games. In: Logic in Computer Science, pp. 275–284 (2006)

[PPS06] Piterman, N., Pnueli, A., Saár, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)

[Zie98] Zielonka, W.: Infinite games on finitely coloured graphs with applications to automata on infinite trees. Theoretical Computer Science 200(1-2), 135–183 (1998)

RATSY– A New Requirements Analysis Tool with Synthesis*

Roderick Bloem[1], Alessandro Cimatti[2], Karin Greimel[1], Georg Hofferek[1], Robert Könighofer[1], Marco Roveri[2], Viktor Schuppan[2], and Richard Seeber[1]

[1] Graz University of Technology, Austria
[2] Fondazione Bruno Kessler, Trento, Italy

Abstract. Formal specifications play an increasingly important role in system design-flows. Yet, they are not always easy to deal with. In this paper we present RATSY, a successor of the Requirements Analysis Tool RAT. RATSY extends RAT in several ways. First, it includes a new graphical user interface to specify system properties as simple Büchi word automata. Second, it can help debug incorrect specifications by means of a game-based approach. Third, it allows correct-by-construction synthesis of systems from their temporal properties. These new features and their seamless integration assist in property-based design processes.

1 Introduction

Several (recent) trends in designing and implementing complex digital systems necessitate the existence of a formal specification for the system at hand. For example, specifications can be used to unambiguously communicate design intents and interface assumptions between collaborating designers. They can also be used to formally verify implementations by means of a model checker. Moreover, a complete formal specification may be used to automatically synthesize an implementation using tools like Lily [6], Anzu [7], or as shown in [11]. Formal specifications are also created, sold, and used as third-party verification IPs [4].

For some of these use cases it is of interest to create and analyze the formal specification stand-alone, i.e., without a corresponding implementation, or before such an implementation is ready. The tool RAT [2] supports these tasks by allowing the user to write a specification in PSL syntax, to analyze it on a trace level, and to check if it is realizable, i.e., if a conforming system exists. However, RAT has some shortcomings when used for system design. Figure 1 depicts a typical property-based design flow. Some informal design intent is turned into a formal specification, which is then refined in several iterations involving simulation and debugging. Finally, an implementation is derived from the specification, ideally using correct-by-construction synthesis. The user faces several problems when putting this design flow into practice. First, it is hard to express the design intent in a formal language. Second, our experience shows

* This work was supported by EU grants 217069 (COCONUT) and 248613 (DIAMOND) as well as Provincia Autonoma di Trento grant EMTELOS.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 425–429, 2010.

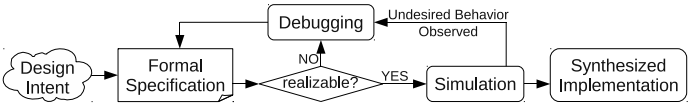


Fig. 1. A typical property-based design flow

that formal specifications for complex designs are hardly ever written correctly on the first try. Thus, there is a need for proper support in debugging. Finally, it must be possible to synthesize an implementation from the specification.

We present the tool RATSY, an extension of RAT which provides several new features to assist the user in a property-based design flow. First, a graphical user interface for drawing Büchi automata has been added. These automata are an easy-to-understand way to specify system properties. Second, the debugging approach presented in [8] has been integrated. It aids in debugging unrealizable specifications and in refining specifications that allow undesired behavior. Third, synthesis functionality has been added. Finally, an additional realizability algorithm [10,5] has been implemented, handling a strictly larger subset of PSL than the one in RAT [9]. Together with the analysis features inherited from RAT, this yields a powerful tool with full support for property-based design processes.

RATSY is available at `http://rat.fbk.eu/ratsy/`. The following sections will detail the improvements and new features of RATSY.

2 Automaton Editor

The automaton editor provides an intuitive interface to specify system properties as Büchi automata. The graphical representation makes creating and especially maintaining specifications easier and less error-prone. Automata are restricted to be deterministic and complete to allow for more efficient synthesis.[1] Completeness is ensured by providing an implicit "dead state" as the default destination of transitions. When transitions are added or changed, other transition conditions are updated by the tool to maintain determinism. Automata can be drawn once and instantiated multiple times, with template parameters allowing for different instantiations. For use with other features of RATSY, PSL formulas are generated automatically from the automata. Finally, the automata are used to visualize state information during simulation and debugging (see next section).

3 Simulation and Debugging

RATSY implements the ideas presented in [8] to test and debug formal specifications. First, it allows the user to test realizable specifications. An implementation is synthesized and the user can simulate it. Second, the tool provides an easy-to-use method to rule out undesired behavior observed during simulation. The user

[1] This is usually not a limiting factor since specifications used in practice tend to be in this class [9]; otherwise the designer can fall back to entering formulas in PSL.

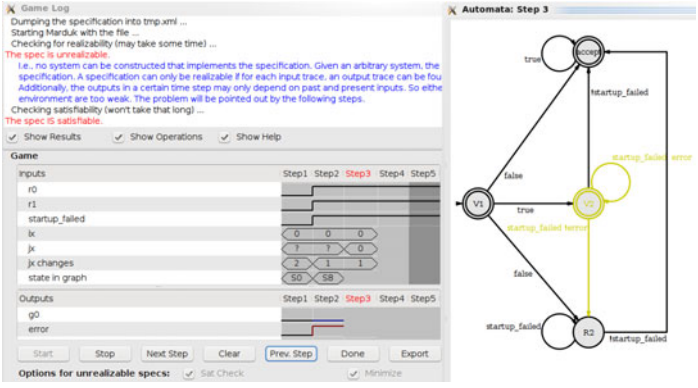


Fig. 2. A part of RATSY's GUI

can simply modify an incorrect simulation trace to match the design intent. The tool then turns this modified trace into a property which enforces the desired response to the inputs, thereby eliminating the undesired behavior. Third, the user can debug unrealizable specifications. Following [8], the user plays a diagnostic game in the role of the system against a counterstrategy or a countertrace synthesized from an unrealizable core of the specification. The diagnostic game illustrates where the specification is too restrictive to be realizable.

Figure 2 shows a part of the screen when playing a diagnostic game to debug unrealizability. Testing a realizable specification looks similar. In every step of the game the counterstrategy determines values for the input signals, and the user sets values for the output signals in the game window or via the automata window. In every automaton of the specification, the current state, as well as transitions that can still be taken, are highlighted. The user can traverse a certain transition by clicking it. All restrictions on signal values associated with that transition are then applied. This integration with the automaton editor greatly increases the usability and helps the user to keep track of what is going on.

4 Technical Aspects

RATSY itself is implemented in Python. The symbolic algorithms rely on CUDD [12] and NuSMV [3], which are accessed through a SWIG-generated [1] wrapper. The synthesis functionality is based on a Python reimplementation of Anzu [7] with some minor implementation-specific improvements. The synthesis algorithm [9] handles specifications given in *Generalized Reactivity (1)* (GR(1)) format. By means of the NuSMV parser, RATSY can perform several syntactic transformations on its own in order to turn a specification into the required format. Furthermore, the NuSMV library automatically encodes multi-valued variables to Boolean signals. RATSY generates circuits in BLIF and Verilog format. If syntactic transformation into GR(1) fails, but succeeds into LTL, then a

preliminary implementation of an algorithm along the lines of [10,5] can be used to determine realizability; debugging and synthesis are not yet available in that case. The conversion into non-deterministic Büchi automata required by [10,5] is performed via a (slightly adapted) version of Wring [13] from Lily [6]. RATSY performs similar to Anzu [7,8] when operating on GR(1) specifications, and can decide most of the examples that Acacia [5] can for full LTL.

5 Conclusions and Future Work

RATSY enhances the analysis features of RAT with a game-based debugging approach for specifications. Furthermore, it eases specifying properties by representing them as Büchi automata, which can be edited via a graphical user interface. Once the user is satisfied with the result of debugging and analyzing her specification, she can synthesize an implementation with just a few clicks. All the new features integrate seamlessly with the well-established analysis features of RAT. Thus, RATSY is a powerful tool to support property-based design.

In the future, we plan to implement a wider variety of output formats for synthesis. Furthermore, we will continue work on improving the size of the synthesized circuits, as well as the time needed to perform synthesis. Concerning debugging, we plan to combine the already implemented approach with model-based diagnosis techniques. This should further simplify the localization of errors.

References

1. Beazley, D.M.: SWIG: An easy to use tool for integrating scripting languages with C and C++. In: Proc. 4th USENIX Tcl/Tk Workshop, pp. 129–139 (1996)
2. 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)
3. 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. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 359–364. Springer, Heidelberg (2002)
4. Dellacherie, S.: Automatic bus-protocol verification using assertions. In: GSPx'04 (2004)
5. Filiot, E., Jin, N., Raskin, J.: An antichain algorithm for LTL realizability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 263–277. Springer, Heidelberg (2009)
6. Jobstmann, B., Bloem, R.: Optimizations for LTL synthesis. In: FMCAD, pp. 117–124 (2006)
7. 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)
8. Könighofer, R., Hofferek, G., Bloem, R.: Debugging formal specifications using simple counterstrategies. In: FMCAD, pp. 152–159 (2009)
9. Piterman, N., Pnueli, A., Saár, Y.: Synthesis of reactive(1) designs. In: Emerson, E.A., Namjoshi, K.S. (eds.) VMCAI 2006. LNCS, vol. 3855, pp. 364–380. Springer, Heidelberg (2005)

10. Schewe, S., Finkbeiner, B.: Bounded synthesis. In: Namjoshi, K.S., Yoneda, T., Higashino, T., Okamura, Y. (eds.) ATVA 2007. LNCS, vol. 4762, pp. 474–488. Springer, Heidelberg (2007)
11. Sohail, S., Somenzi, F.: Safety first: A two-stage algorithm for LTL games. In: FMCAD, pp. 77–84 (2009)
12. Somenzi, F.: CUDD: CU Decision Diagram Package. University of Colorado at Boulder, ftp://vlsi.colorado.edu/pub/
13. 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)

Comfusy: A Tool for Complete Functional Synthesis
(Tool Presentation)

Viktor Kuncak, Mikaël Mayer, Ruzica Piskac, and Philippe Suter*

Swiss Federal Institute of Technology (EPFL), Switzerland
firstname.lastname@epfl.ch

Abstract. Synthesis of program fragments from specifications can make programs easier to write and easier to reason about. We present Comfusy, a tool that extends the compiler for the general-purpose programming language Scala with (non-reactive) functional synthesis over unbounded domains. Comfusy accepts expressions with input and output variables specifying relations on integers and sets. Comfusy symbolically computes the precise domain for the given relation and generates the function from inputs to outputs. The outputs are guaranteed to satisfy the relation whenever the inputs belong to the relation domain. The core of our synthesis algorithm is an extension of quantifier elimination that generates programs to compute witnesses for eliminated variables. We present examples that demonstrate software synthesis using Comfusy and illustrate how synthesis simplifies software development.

1 Introduction

Synthesis is among the most ambitious techniques for building correct computer systems [4]. Recently, we have seen advances of synthesis for finite-state reactive systems [6,1]. In this paper, we describe a step in another direction: synthesis for *infinite-state* non-reactive software systems [2]. Our goal is to gradually introduce synthesis into software development by supporting new programming language constructs that leverage synthesis in delimited portions of the program. Specifically, we introduce a programming language construct, choose. The choose construct accepts a parameterized predicate P. It synthesizes a function that maps the parameters to output values satisfying P. We restrict the language of predicates to a decidable logic, and provide a complete synthesis procedure: whenever a value satisfying the predicate exists, the synthesized function will compute one such value.

We continue by illustrating our system through examples. We then define our synthesis problem more precisely and describe our implementation.[1]

* The author list has been sorted according to the alphabetical order; this should not be used to determine the extent of authors' contributions. Ruzica Piskac was supported in part by the SNF Grant SCOPES IZ73Z0_127979. Philippe Suter was supported by the SNF Grant 200021_120433.

[1] For further details, see [2] and http://lara.epfl.ch/dokuwiki/comfusy

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 430–433, 2010.

2 Examples

Linear arithmetic. As a first example, consider the problem of decomposing a number of seconds into hours, minutes and the leftover seconds. We can specify this problem as follows:

```
val (hours, minutes, seconds) = choose((h: Int, m: Int, s: Int) ⇒ (
    h ∗ 3600 + m ∗ 60 + s == totsec && 0 ≤ m && m < 60 && 0 ≤ s && s < 60))
```

On this example, Comfusy generates the following code:[2]

```
val (hours, minutes, seconds) = {
  val loc1 = totsec div 3600
  val num2 = totsec + ((−3600) ∗ loc1)
  val loc2 = min(num2 div 60, 59)
  val loc3 = totsec + ((−3600) ∗ loc1) + (−60 ∗ loc2)
  (loc1, loc2, loc3)
}
```

Arithmetic pattern matching. We also found synthesis for linear arithmetic to be useful for extending pattern-matching in a way that is similar to, but goes beyond Haskell $(n+k)$-patterns. The following code implements the fast exponentiation algorithm:

```
def pow(base : Int, p : Int) = {
  def fp(m : Int, b : Int, i : Int) = i match {
    case 0 ⇒ m
    case 2∗j ⇒ fp(m, b∗b, j)
    case 2∗j+1 ⇒ fp(m∗b, b∗b, j)
  }
  fp(1,base,p)
}
```

The third pattern, for instance, will match the integer i if there exists an integer j such that i == 2 * j + 1. The pattern also works as a binder, and the value computed for j is thus available on the right hand side. Comfusy checks that the match expression is exhaustive and that no pattern is subsumed by the previous ones, and emits a warning if it can find a value matched by no pattern or if a pattern in unreachable.

Parametrized linear arithmetic. The previous two examples are in standard linear arithmetic. Comfusy can also handle constraints expressed in *parametrized* linear arithmetic, that is, constraints that are not linear at compile-time but become linear at run-time, when some of the values are known. For example, the following code computes, if it exists, the integer ratio between two numbers a and b:

```
val ratio = choose((r: Int) ⇒ a == r ∗ b || b == r ∗ a)
```

[2] The div operator computes the floored integer division. For example -1 div $2 = -1$.

Although the term r ∗ b, for instance, is not linear at compile-time, the value of b is known at run-time at the point where the value of r needs to be computed. The synthesized code thus needs to handle all possible values of the parameters a and b.

Set constraints. Finally, Comfusy can be used to synthesize code handling sets. Consider the following example:

```
val (a1,a2) = choose((a1:Set[O],a2:Set[O]) ⇒
      a1 ++ a2 == s && a1 ** a2 == Set.empty
  && a1.size − a2.size ≤ 1 && a2.size − a1.size ≤ 1)
```

Here, ++ and ** denote set union and intersection respectively. The generated code constructs two sets a1 and a2 such that they form a partition of the existing set s, with the additional constraint that the sizes of a and b should not differ by more that 1. Note that requiring that their sizes be identical would result in an unsatisfiable set of constraints whenever the size of s is odd.

3 Definition and Algorithm for Synthesis in Comfusy

Definitions. Let $\mathsf{FV}(q)$ denotes the set of free variables in a formula or term q. If $\boldsymbol{x} = (x_1, \ldots, x_n)$ then $\boldsymbol{x}_s$ denotes the set of variables $\{x_1, \ldots, x_n\}$. If q is a term or formula, $\boldsymbol{x} = (x_1, \ldots, x_n)$ a vector of variables and $\boldsymbol{t} = (t_1, \ldots, t_n)$ a vector of terms, then $q[\boldsymbol{x} := \boldsymbol{t}]$ denotes the term resulting from substituting in q free variables $x_1, \ldots, x_n$ with terms $t_1, \ldots, t_n$, respectively.

Definition 1 (Synthesis Procedure). *A synthesis procedure takes as input a formula F and a vector of variables $\boldsymbol{x}$ and outputs a pair of*

1. *a precondition formula* pre *with* $\mathsf{FV}(\mathsf{pre}) \subseteq \mathsf{FV}(F) \setminus \boldsymbol{x}_s$
2. *a tuple of terms $\boldsymbol{\Psi}$ with* $\mathsf{FV}(\boldsymbol{\Psi}) \subseteq \mathsf{FV}(F) \setminus \boldsymbol{x}_s$

such that the following two implications are valid:

$$\exists \boldsymbol{x}.F \rightarrow \mathsf{pre}$$
$$\mathsf{pre} \rightarrow F[\boldsymbol{x} := \boldsymbol{\Psi}]$$

Algorithms. Our core specification language is quantifier-free Boolean Algebra with Presburger Arithmetic (BAPA) [3].[3] Our procedure for integer linear arithmetic synthesis is related to the Omega-test algorithm [7]. One of the key differences is that our procedure computes witness terms for eliminated variables. Additionally, in the parametrized arithmetic case, some choices in the algorithm need to be delayed until the run-time values are known; the synthesized code must account for these choices by generating different cases for different signs of coefficients and by, e.g., invoking a GCD algorithm in the generated code. The algorithm for constraints on sets is based on a witness-generating version of [3].

[3] We currently do not support quantifiers in the specification predicates. Quantifiers do not increase the set of definable relations, because BAPA has quantifier elimination [3]. We could support quantifiers by running the quantifier elimination algorithm first, then invoking our synthesis procedure.

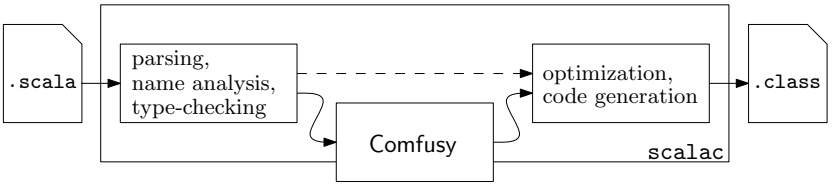


Fig. 1. Interaction of Comfusy with `scalac`, the Scala compiler. Comfusy takes as an input the abstract syntax tree of a Scala program and rewrites calls to choose to syntax trees representing the synthesized function.

4 Implementation

We have implemented Comfusy as a plugin for the Scala compiler (scalac), adding a phase to the standard compilation process (see Figure 1). During this phase, our plugin extracts calls to the choose function and arithmetic patterns and replaces them by code that computes the appropriate values. The input and output of Comfusy are thus abstract syntax trees in the internal format of scalac. The compiler then proceeds as usual, so all further optimizations are applied to the synthesized code as well. Comfusy supports synthesis for predicates expressed in integer linear arithmetic, parametrized linear arithmetic, and set algebra with size constraints, as well as linear arithmetic patterns. Comfusy can also check whether the synthesis predicates are always satisfiable (for all possible run-time values of the program variables) or whether they describe unique solutions, and emit compile-time warnings with counter-examples when necessary. We use an off-the-shelf decision procedure for these checks [5]. In our experience, the execution time of the synthesized code is similar to equivalent hand-written code. We also found the compile-time overhead to be negligible.

References

1. 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)
2. Kuncak, V., Mayer, M., Piskac, R., Suter, P.: Complete functional synthesis. In: ACM Conf. Programming Language Design and Implementation, PLDI (2010)
3. Kuncak, V., Nguyen, H.H., Rinard, M.: Deciding Boolean Algebra with Presburger Arithmetic. Journal of Automated Reasoning 36(3), 213–239 (2006)
4. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Communications of the ACM 14(3), 151–165 (1971)
5. 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)
6. Pnueli, A., Rosner, R.: On the synthesis of a reactive module. In: ACM Symp. Principles of Programming Languages, POPL (1989)
7. Pugh, W.: A practical algorithm for exact array dependence analysis. Communications of the ACM 35(8), 102–114 (1992)

Universal Causality Graphs: A Precise Happens-Before Model for Detecting Bugs in Concurrent Programs

Vineet Kahlon and Chao Wang

NEC Laboratories America, Princeton, NJ 08540, USA

Abstract. Triggering errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of operations of different threads. Efficient static techniques, therefore, play a critical role in restricting the set of interleavings that need be explored in greater depth. The goal here is to exploit scheduling constraints imposed by synchronization primitives to determine whether the property at hand can be violated and report schedules that may lead to such a violation. Towards that end, we propose the new notion of a *Universal Causality Graph (UCG)* that given a correctness property P, encodes the set of all (statically) feasible interleavings that may violate P. UCGs provide a unified happens-before model by capturing causality constraints imposed by the property at hand as well as scheduling constraints imposed by synchronization primitives as causality constraints. Embedding all these constraints into one common framework allows us to exploit the synergy between the constraints imposed by different synchronization primitives, as well as between the constraints imposed by the property and the primitives. This often leads to the removal of significantly more redundant interleavings than would otherwise be possible. Importantly, it also guarantees both soundness and completeness of our technique for identifying statically feasible interleavings. As an application, we demonstrate the use of UCGs in enhancing the precision and scalability of predictive analysis in the context of runtime verification of concurrent programs.

1 Introduction

Detecting errors in concurrent programs is a notoriously difficult task. A key reason for this is the behavioral complexity resulting from the large number of interleavings of different threads. This leads to the state-explosion problem which renders a full-fledged state space exploration of concurrent programs infeasible. As a result, in recent years runtime error detection techniques have been gaining in popularity. These come in many variants. Runtime monitoring aims at identifying violations exposed by a given execution trace [12, 18, 5, 9]. However, due to the large number of interleavings of the program, triggering a concurrency bug by exploring just one interleaving is unlikely. In contrast, runtime prediction aims at detecting violations in all feasible interleavings of events of the given trace. In other words, even if no violation exists in that trace, but an alternative interleaving is erroneous, a predictive method [8, 15, 2, 7, 6, 14] may be able to catch it without actually re-running the test.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 434–449, 2010.

Predictive analysis seems to offer a good compromise between runtime monitoring and full-fledged model checking in that it guarantees better coverage than runtime monitoring but mitigates the state explosion inherent in model checking. In its most general form, predictive analysis has three steps (1) Run a test of the concurrent program to obtain an execution trace. (2) Run a sound (over-approximate) static analysis of the given trace to detect *potential* violations, e.g., data races, atomicity violations, etc. If no violation is found, return. (3) Build a precise predictive model, and for each potential violation, check whether it is feasible. If it is feasible, find a concrete and replayable witness trace. Many variants of this basic framework have been proposed in the literature to explore the various tradeoffs between scalability and precision. Clearly, the main bottleneck in scalability is the feasibility check in step 3 (essentially model checking).

In the interest of scalability some techniques avoid step 3 altogether. For instance, Farzan et. al. [6] have proposed a static analysis for predicting atomicity violations in which they focus on the control paths and model only nested locks. For threads synchronizing via nested locks only and assuming no data variables, their analysis is sound and complete, in that step 3 can be avoided. However, this technique is not applicable to programs using non-nested locks or synchronization primitives other than locks, including wait/notify, barriers, etc. As a result, the reported violations may be spurious. Although such warnings can serve as hints for subsequent analysis, they are not immediately useful to programmers because deciding whether they are real errors remains a challenging task. Other techniques try to address the scalability problem by exploring only a small subset of the feasible interleavings via trace-based under-approximations [15, 2, 14] thereby suffering from a very limited coverage of interleavings.

If precision is of paramount concern then static analysis (step 2) is augmented with model checking in step 3 wherein the feasibility of the set of statically generated warnings can be verified. Since model checking is computationally expensive, it is imperative that the static analysis be made as precise as possible. First, if static analysis can deduce that a set of warning locations is simply unreachable then the expensive step 3 can be avoided altogether. Second, if static analysis can deduce invariants with respect to the trace and the property at hand, we can use them to weed out many interleavings that need be explored via step 3 thereby enhancing its scalability. Therefore, irrespective of the predictive analysis method being used, step 2, i.e., statically detecting potentially erroneous interleavings of events of the given trace, occupies a key role in determining the scalability and precision of the overall framework.

However, existing static analysis techniques suffer from several drawbacks. **1. Comprehensive Handling of Synchronizations:** State-of-the-art static analysis techniques, e.g. [7], apply only to programs with nested locks and are therefore not applicable to programs using non-nested locks or wait/notify-style primitives in conjunction with locks (nested or non-nested) which are very common in Java programs. The mover-based atomicity checkers, such as *atomizer* [8], are conservative even for control path reachability (no data); they typically can robustly handle locks but not the other synchronization primitives. In contrast, our new static analysis technique can handle multiple synchronization primitives used in real-life programs (e.g. Java) in a unified manner, and is both sound and complete for control path reachability for two threads, i.e., when there is no data, or data does not affect the control flow of the program. **2. Causal**

Constraints from Properties: The existence of standard concurrency errors like data races and atomicity violations can be expressed uniformly as a set of happens-before constraints between events of different threads. These property-induced constraints can be used in conjunction with scheduling constraints imposed by synchronization primitives to infer yet more happens-before constraints. To our knowledge, the interaction of these two types of constraints has not been exploited by existing techniques which therefore end up retaining more (spurious) interleavings than are necessary.

The main contribution of this paper is the new notion of a *Universal Causality Graph (UCG)*, which is a unified happens-before model for the given (trace) program as well as the property at hand, that addresses the above challenges. UCGs allow us to capture, as happens-before constraints, the set of all possible interleavings that are feasible under the scheduling constraints imposed by synchronization primitives that may potentially lead to violations of the property at hand. With a given execution trace of a program specified as a set of local computations $x^1, ...x^n$ of n threads and a property P, we associate a UCG, denoted by $U_{(x^1,...,x^n)}(P)$, which is a directed graph whose vertices are a subset of the set of synchronization events occurring along $x^1, ..., x^n$ and each of whose edges $e_1 \rightsquigarrow e_2$, represents a happens-before constraint, i.e., e_1 must be executed before e_2. Thus the UCG implicitly captures the set of all interleavings of $x^1, ..., x^n$ that satisfy all the happens-before constraints represented by its edges.

UCGs have the following desirable properties **(a) Precision:** If data is not tracked, for two threads the UCG captures precisely the set of interleavings of $x^1, \ldots, x^2$ satisfying the property P, e.g., the existence of a data race or atomicity violation. For an arbitrary number of threads the set of interleavings captured is a super-set of the set of interleavings satisfying P. In other words, the analysis is sound in general and complete for two threads interacting via synchronization primitives only. **(b) Unified View:** The UCG encodes both property-induced causality constraints and scheduling constraints imposed by synchronization primitives in terms of happens-before constraints. Unlike existing techniques, it can handle multiple synchronization primitives in a unified manner. This enables us to leverage the synergy between causality constraints induced by both the property as well as the program, and allows us to deduce more causal constraints than would otherwise be possible. More importantly, these constraints are necessary to guarantee both soundness and completeness of our method for two threads. **(c) Scalability:** Since the given trace could be arbitrarily long, incorporating all synchronization events of the trace as vertices and all the deduced causal constraints as edges into the UCG would impact the scalability of the analysis. However, we show that, for predictive analysis of a given property, the UCG need not keep track of causality edges induced by the entire traces $x^1, ..., x^n$ but only short suffixes thereof. The novelty of this *chopping result* lies in the existence of a set of special *lock-free control states*, from which the UCG based analysis can still guarantee both soundness and completeness.

UCGs are a generalization of lock causality graphs (LCGs) [10] which were formulated to reason about pairwise reachability for threads communicating purely via locks. However, LCGs could only be used to reason about programs using locks. Furthermore, LCGs were formulated to reason only about pairwise reachability and therefore could not exploit causality constraints induced by properties such as atomicity violations. Also, our UCG based analysis is a backward inference process starting from the

property at hand—it does not enumerate interleavings. This differs from the forward analysis used in [15, 2, 14] which explicitly enumerate thread interleavings.

Happens-before constraints have been exploited before for predictive analysis for detecting races and atomicity violations [4, 9]. However, the causal models considered were restrictive in that the set of interleavings explored had to preserve the global ordering of lock/unlock statements in the original global computation x. Since the number of such interleavings is a small fraction of the total number of feasible interleavings of local computations of different threads along x, these techniques could miss detection of errors and were therefore not guaranteed sound (though they were sound with respect to the global ordering of lock/unlock statements along x). UCGs, on the other hand, allow the lock/unlock statements from different threads to be re-ordered relative to each other and will therefore explore all possible statically feasible interleavings of local computations of different threads along x. This guarantees soundness of our technique.

The proofs of all the results can be found in the full version available on-line [1].

2 Preliminaries

A *concurrent program* has a set $\{T_1, \ldots, T_n\}$ of threads and a set SV of *shared variables*. Each thread T_i, where $1 \leq i \leq n$, has a set of *local variables* LV_i. Let $Tid = \{1, \ldots, n\}$ be the set of thread indices. Let $V_i = SV \cup LV_i$, where $1 \leq i \leq n$, be the set of variables accessible in T_i. An execution trace of the program is a sequence $x = t_1 \ldots t_K$ of events. An *event* $t \in x$ is a tuple $\langle tid, action \rangle$, where $tid \in Tid$ and $action$ is of one of the form (let $tid = i$)

- guarded assignment $(\mathsf{assume}(g), asgn)$, where g is a condition over V_i and $asgn$ is a set of assignments, each of the form $v := exp$, where $v \in V_i$ and exp is an expression over V_i. Intuitively, g must be true for the assignments to proceed.
- $fork(j)$ and $join(j)$. The former models the creation of child thread T_j by thread T_i. The latter models that thread T_i waits for thread T_j to join back.
- $lock(l)$ and $unlock(l)$. The former models the acquire of lock l. The latter models the release of lock l.
- $wait_{pre}(c)$, $wait_{post}(c)$ and $notify(c)$. The first two, when combined, model the wait of condition variable c. The last event models the notification of c.

Each event t in x is a unique execution instance of a statement in the program. If a statement in the program is executed multiple times, e.g., in a loop or a recursive function, each execution instance is modeled as a separate event. If we project x back to the local threads, each current thread $x^1, \ldots, x^n$ is a purely straight-line program.

Synchronization Primitives. In both Java and PThreads, the primitive `wait(c,l)`, where l is a lock and c is a condition variable, is a composite statement. Before calling it, thread T_i is expected to hold lock l. Upon calling it, thread T_i releases lock l, and then blocks—waiting for another thread T_j to call `notify(c)`. After that, and only when lock l is available again, thread T_i wakes up and immediately re-acquires lock l. Therefore, for verification purposes, we model `wait(c,l)` using the semantically equivalent event sequence $wait_{pre}(c)$; $unlock(l)$; $lock(l)$; $wait_{post}(c)$. Also note that

the suggested way of using condition variables, in both Java and PThreads, is to wrap both `wait(c,l)` and `notify(c)` with a pair of `lock(l)` and `unlock(l)`.

By defining the expression syntax suitably, the guarded assignment event itself is expressive enough to model the execution of all kinds of statements including synchronization primitives. In fact, this is what we have implemented in the model checking procedure at the final step. The reason why we represent lock-unlock events and wait-notify events separately is for convenience in understanding the UCG-based static analysis; in static analysis, our focus is on these concurrency/synchronization events only (data is ignored). To understand the expressiveness of guarded assignment, consider the following variants: (1) when the guard is true, the set $asgn$ models normal assignment statements; (2) when the set $asgn$ is empty, $\mathsf{assume}(g)$ models a branching statement `if(cond)`; and (3) with both the guard and the assignment set, it can model the atomic *check-and-set* operation, which is the foundation of all synchronization primitives. For example, acquire of lock l in thread T_i, where $i \in Tid$, is modeled as event $\langle i, (\mathsf{assume}(l = 0), \{l := i\})\rangle$; here 0 means the lock is available and thread index i indicates the lock owner. Release of lock l is modeled as $\langle i, (\mathsf{assume}(l = i), \{l := 0\})\rangle$.

Concurrent Trace Programs. The semantics of an execution trace $x = e_1 \dots e_K$ is defined using a state transition system. Let V be the set of all program variables and Val be a set of values of variables in V. A *state* is a map $s : V \rightarrow Val$ assigning a value to each variable. We also use $s[v]$ and $s[exp]$ to denote the values of $v \in V$ and expression exp in state s. We say that a *state transition* $s \xrightarrow{t} s'$ exists, where s, s' are states and e is a guarded assignment event in thread T_i, if the action is $(\mathsf{assume}(g), asgn)$, $s[g]$ is true, and for each assignment $v := exp$ in $asgn$, $s'[v] = s[exp]$ holds; states s and s' agree on all other variables. The execution trace $x = t_1 \dots t_K$ can be viewed as a total order of the events along x. From x one can derive a partial order called the concurrent trace program (CTP) [17].

Definition 1. *The* concurrent trace program *with respect to x, denoted CTP_x, is a partially ordered set $(T, \sqsubseteq)$ such that, $T = \{t \mid t \in x\}$ is the set of events, and for any $t_i, t_j \in T$, $t_i \sqsubseteq t_j$ iff*

- *$tid(t_i) = tid(t_j)$ and $i < j$; or*
- *t_i has action $fork(tid(t_j))$; or*
- *t_j has action $join(tid(t_i))$; or*
- *t_i has action $wait_{pre}(c)$ and t_j has the matching $notify(c)$; or*
- *t_j has action $wait_{post}(c)$ and t_i has the matching $notify(c)$.*

Intuitively, CTP_x orders events from the same thread by their execution order along x, and orders events from different threads by the causal relations of fork-join and wait-notify. Otherwise, events from different threads are not *explicitly* ordered with respect to each other.

We now define *feasible linearizations* of CTP_x. Let $x' = t'_1 \dots t'_n$ be a linearization of CTP_x, i.e. an interleaving of events of x. We say that x' is *feasible* iff there exist states $s_0, \dots, s_n$ such that, s_0 is the initial state of the program and for all $i = 1, \dots, n$, there exists a transition $s_{i-1} \xrightarrow{t'_i} s_i$. This definition captures the standard sequential

consistency semantics for concurrent programs, where we modeled concurrency primitives such as locks by using auxiliary shared variables.

Causal Models for Feasible Linearizations. We recall that in predictive analysis the given concurrent program is first executed to obtain an execution trace x. By projecting x onto the local states of individual threads one can obtain CTP_x. Then given a property P, e.g., existence of data races or atomicity violations, the goal of predictive analysis is to find a feasible linearization of CTP_x that satisfies P.

Since the total number of linearizations of CTP_x may be too large, *static* analysis is often employed to isolate a (small) set of linearizations of CTP_x whose feasibility can then be checked via model checking. Here static feasibility implies that data is typically ignored and the linearizations generated are required to be feasible only under the scheduling constraints imposed by synchronization and fork-join primitives. We propose the notion of a *Universal Causality Graph* that captures precisely the set of feasible interleavings of CTP_x that may lead to violations while guaranteeing (i) soundness in general and completeness for two threads, (ii) scalability, (iii) handling of different synchronization primitives in a unified manner, and (iv) exploiting the synergy between causal constraints imposed by the property as well as the program. To the best of our knowledge, none of the existing techniques satisfies all four of these requirements.

3 Universal Causality Graph

Given a set of local computations $x^1, ..., x^n$ of threads T_1,..., T_n, respectively, and a standard property P such as the presence of a data race or an atomicity violation, we construct a causality graph, denoted $U_{(x^1,...,x^n)}(P)$, such that there exists an interleaving of $x^1, ..., x^n$ satisfying P if and only if $U_{(x^1,...,x^n)}(P)$ is acyclic. We express both the property as well as scheduling constraints imposed by synchronization primitives in terms of happens-before constraints. To start with, we show how to express the occurrence of a property violation as a set of happens-before constraints.

3.1 Properties as Causality Constraints

We consider two standard concurrency violations: data races and atomicity violations.

Data Races. A data race occurs if there exist events t_a and t_b of two different threads such that (a) a common shared variable is accessed by t_a and t_b with at least one of the accesses being a write operation, and (b) there exists a reachable (global) state of the concurrent program in which both t_a and t_b are enabled. In order to express the occurrence of a data race involving t_a and t_b, we introduce the two happens-before constraints $t_{a'} \rightsquigarrow t_b$ and $t_{b'} \rightsquigarrow t_a$ in the universal causality graph, where $t_{a'}$ and $t_{b'}$ are the events immediately preceding t_a and t_b in their respective threads. Note that given an execution trace, $t_{a'}$ and $t_{b'}$ are defined uniquely.

Atomicity Violations. A three-access atomicity violation [12, 6, 17] involves an event sequence $t_c \ldots t_r \ldots t_{c'}$ such that (a) t_c and $t_{c'}$ are in a user transaction of one thread, and t_r is in another thread, and (b) t_c and t_r are data dependent; and t_r and $t_{c'}$ are data dependent. Depending on whether each event is a *read* or *write*, there are eight possible combinations of the triplet $t_c, t_r, t_{c'}$. While R-R-R, R-R-W, and W-R-R are

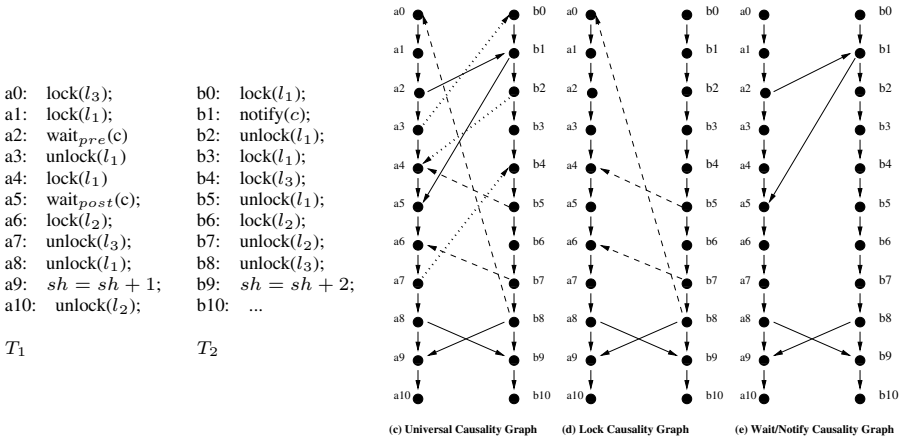


Fig. 1. An Example Universal Causality Graph

serializable, the remaining five may indicate atomicity violations. Given the CTP_x and a transaction $trans = t_i \ldots t_j$, where $t_i \ldots t_j$ are events from a thread in x, the set of potential atomicity violations can be computed by scanning the trace x once, and for each remote event $t_r \in CTP_x$, finding the two local events $t_c, t_{c'} \in trans$ such that $\langle t_c, t_r, t_{c'} \rangle$ forms a non-serializable pattern. Such an atomicity violation can be captured via the two happens-before constraints $t_c \rightsquigarrow t_r$ and $t_r \rightsquigarrow t_{c'}$.

3.2 Universal Causality Graph Construction

We motivate the concept of a *Universal Causality Graph (UCG)* via an example comprised of local traces x^1 and x^2 of threads T_1 and T_2, respectively, shown in fig 1. Suppose that we are interested in deciding whether *a9* and *b9* constitute a data race. Since the set of locks held at *a9* and *b9* are disjoint, these pair of locations constitute a potential data race. Furthermore, since the traces use wait/notify statements as well as non-nested locks, we cannot use existing techniques [7, 6, 11] to decide simultaneous reachability of *a9* and *b9*. As discussed in Sec. 3.1, for the race to occur there must exist an interleaving of x^1 and x^2 that satisfies the constraints $a8 \rightsquigarrow b9$ and $b8 \rightsquigarrow a9$. Furthermore, the locks along x^1, x^2 must be acquired in a consistent fashion and causality constraints imposed by wait/notify events must be respected.

We now show that the causality constraints generated in the UCG by the property as well as scheduling constraints imposed by locks, fork/join, and wait/notify events that are *relevant* in exposing the data race, can be captured in a unified manner. For two arbitrary events c_1 and c_2 of $U_{(x^1,x^2)}(P)$, there exists an edge $c_1 \rightsquigarrow c_2$ if c_1 must be executed before c_2 in order for P to hold.

A UCG has two types of edges (i) *Seed* edges and (ii) *Induced* edges. Seed edges, shown as bold edges in the UCG in Fig. 1(c), can be further classified as *Property* and *Synchronization* seed edges.

Property Seed Edges are introduced by properties as in Sec. 3.1. In our example, the potential data race at *a9* and *b9* introduces the edges $a8 \rightsquigarrow b9$ and $b8 \rightsquigarrow a9$.

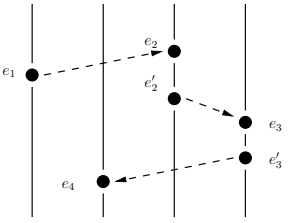


Fig. 2. A Causality Chain from e_1 to e_4

Synchronization Seed Edges are induced by fork/join and the various synchronization primitives. For simplicity, we only discuss wait/notify. Locks do not add seed edges.

- **Wait-Notify:** Recall that the primitive $wait(c, l_1)$ is modeled as the event sequence $a1 : lock(l_1)$, $a2 : wait_{pre}(c)$, $a3 : unlock(l_1)$, $a4 : lock(l_1)$, $a5 : wait_{post}(c)$. Let $b1$ be the matching $notify(c)$. The seed edges are $a2 \rightsquigarrow b1$ and $b1 \rightsquigarrow a5$. Note that since x^1 and x^2 are generated from concrete traces for each notify statement there exists a unique matching wait statement, if any, and vice versa. Strictly speaking, we should consider all scenarios wherein notify statements are executed without the matching wait statements. However, that will block the wait statements causing us to miss potential violations. In order to maximize the number of violations, we assume that all pairs of matching wait/notify statements are executed in unison.
- **Fork-Join:** The first edge is from *fork* to the first event of the child thread. The second edge is from the last event of the child thread to *join*.

The interaction between these seed edges and locks can be used to deduce more constraints that are captured as *induced* edges. They are the dashed edges in Fig. 1(c). These induced edges are key in guaranteeing soundness and completeness.

Induced Edges: Consider the seed causality constraint *b8* $\rightsquigarrow$ *a9*. From this we can deduce the new causality constraint $b7 \rightsquigarrow a6$. Towards that end, we observe that at location *a9*, lock l_2 is held which was acquired at *a6*. Also, once l_2 is acquired at *a6* it is not released until after T_1 exits *a10*. Furthermore, we observe that *b6* is the last statement to acquire l_2 before *b8* and *b7* is its matching release. Due to constraint *b8* $\rightsquigarrow$ *a9* and the local constraint *b7* $\rightsquigarrow$ *b8*, one can deduce, via transitivity, that *b7* $\rightsquigarrow$ *a9*. Moreover, from the mutual exclusion semantics of lock l_2, we have that since l_2 is held at *a9* it must first be released by T_2 before T_1 can acquire it via *a6* without which $a9$ cannot be executed. Thus *a6* must also be executed after *b7*.

From $b7 \rightsquigarrow a6$ one can, in turn, deduce that $b8 \rightsquigarrow a0$. This is because the last statement to acquire l_3 before $b7$ is *b4* and its matching release is *b8*. Using the same argument as above, from the causality constraint *b7* $\rightsquigarrow$ *a6* and mutual exclusion semantics of lock l_3, we can deduce that l_3, which is held at b7, must first be released by T_2 before T_1 can acquire it via *a0* which it needs to in order to execute *a6*, i.e., $b8 \rightsquigarrow a0$. The process is continued till a fixpoint is reached. Fig. 1(b) shows all the induced edges added by starting at the seed edges *b8* $\rightsquigarrow$ *a9* and *a8* $\rightsquigarrow$ *b9*.

Similarly it can be seen that the wait/notify seed edges $a2 \rightsquigarrow b1$ and $b1 \rightsquigarrow a5$ add further induced edges which are not shown for reasons of clarity.

Algorithm 1. Computing the Universal Causality Graph

1: **Input:** Property P and local paths x^1 and x^2 of T_1 and T_2, respectively.
2: Initialize the vertices and edges of $U_{(x^1,x^2)}(P)$ to $\emptyset$
3: Add causality edges for P as defined in sec. 3.1 (**Property Seed Edge**)
4: Add fork-join induced causality edges (**Fork-Join Seed Edge**)
5: **for** each pair of locations w and n corresponding to matching wait/notify events **do**
6: Add edges $w_{pre} \rightsquigarrow n$ and $n \rightsquigarrow w_{post}$, (**Wait/Notify Seed Edge**)
7: **end for**
8: **repeat**
9: **for** each lock l **do**
10: **for** each edge $d_j \rightsquigarrow d_i$ between events d_j and d_i of T_j and T_i, respectively **do**
11: If l is held at d_j and not released after d_j along x^j then add an edge $r_i \rightsquigarrow a_j$, where a_j is the last statement to acquire l before d_j and r_i is the last statement to release l before d_i
12: If l is held at d_j and not released after d_j along x^j and l is held at d_i then output P **does not hold** and **Quit**
13: Let a_j be the last statement to acquire l before d_j along x^j and r_j the matching release for a_j; and let r_i be the first statement to release l after d_i along x^i and a_i the matching acquire for r_i
14: **if** l is held at either d_i or d_j **then**
15: add edge $r_j \rightsquigarrow a_i$ (**Induced Edge**)
16: **end if**
17: **end for**
18: **end for**
19: **until** no new edges can be added
20: **for** $i \in [1..2]$ **do**
21: Add edges among all events of x^i occurring in $U_{(x^1,x^2)}(P)$ to preserve their relative ordering along x^i
22: **end for**

3.3 Computing the Universal Causality Graph

The procedure to compute the UCG is formulated as alg. 1. It adds causality constraints one-by-one (seed edges via steps 3-7, and induced edges via steps 8-19) till it reaches a fixpoint. Note that steps 20-22, preserve the local causality constraints along $x^1, ..., x^n$.

Since each edge in $U_{(x^1,x^2)}(P)$ is a *happens-before* constraint, we see that in order for P to hold $U_{(x^1,x^2)}(P)$ has to be acyclic. In fact, it turns out that for two threads acyclicity is also a sufficient condition leading to the following *Acyclicity Result*.

Theorem 1. *Property P is violated via a (statically) feasible interleaving of local paths x^1 and x^2 of T_1 and T_2, respectively, if and only if $U_{(x^1,x^2)}(P)$ is acyclic.*

3.4 Generalization to n Threads

For the case of n threads, the only modification that is required to alg. 1 is in step 10. Here a causality relation between events d_i and d_j can be induced not only via a single edge of the form $d_j \rightsquigarrow d_i$ but also via a *causality chain* from d_j to d_i (see fig. 2), i.e.,

a sequence of pre-existing causality edges of the form $e_1 \rightsquigarrow e_2$, $e'_2 \rightsquigarrow e_3$, $e'_3 \rightsquigarrow e_4$,..., $e'_{k-1} \rightsquigarrow e_k$, where (i) $e_1 = d_j$ and $e_k = d_i$, and (ii) for each m, e'_m occurs after e_m along $x^{m'}$ for some $m' \in [1..n]$.

Thus the condition at line 10 of alg. 1 is modified as follows:

for each pair of events d_i and d_j belonging to different threads T_i and T_j, respectively, such that there is a causality chain from d_j to d_i **do**

Complexity of the UCG Construction. The total time taken for building the UCG is $O(|E||L|)$, where $|E|$ denotes the number of edges that can be added to the UCG and $|L|$ is the number of different locks acquired/released along $x^1, ..., x^n$. In the worst case $|E|$ is $O((n|N|)^2)$, where $|N|$ is the maximum number of synchronization events occurring along any of the local sequences $x^1, ..., x^n$.

Exploiting the Synergy between Synchronization Primitives. Existing static techniques for reasoning about programs with multiple synchronization primitives like locks and wait/notify consider the scheduling constraints imposed by them separately. Thus a violation is said to exist if it can occur either under scheduling constraints imposed by locks or under those imposed by wait/notifies. However, UCGs capture constraints imposed by all the primitives in a unified manner thereby allowing us to exploit the synergy between them. In our example, by considering constraints imposed by locks and wait/notifies separately we cannot deduce that *a9* and *b9* do not race. Indeed, the scheduling constraints imposed only by locks results in the *acyclic* lock causality graph Fig. 1(d). Similarly, the scheduling constraints imposed only by wait/notify results in the *acyclic* wait/notify causality graph in Fig. 1 (e). In order generate a cycle that proves infeasibility of the data race we have to consider both the primitives in unison. Towards that end, we construct the UCG shown in Fig. 1(c) which has the cycle $a0 \rightsquigarrow a2 \rightsquigarrow b1 \rightsquigarrow b8 \rightsquigarrow a0$ thereby allowing us to deduce that *a9* and *b9* do not constitute a data race.

Exploiting the Synergy between Program and Property. Consider the cycle $a0 \rightsquigarrow a2 \rightsquigarrow b1 \rightsquigarrow b8 \rightsquigarrow a0$ in Fig. 1(c). It is comprised of the induced edge $b8 \rightsquigarrow a0$ and the wait/notify seed edge $a2 \rightsquigarrow b1$. The induced edge $b8 \rightsquigarrow a0$ was added via an induction sequence (via steps 8-18) starting at the property seed edge $b8 \rightsquigarrow a9$. Thus in order to rule out the data race we have to consider the causality constraints induced by the property as well as the synchronization primitives in unison. In contrast, existing techniques do not exploit the synergy between these constraints and are therefore not guaranteed complete for two threads.

3.5 Handling Multiple Properties

For clarity, the UCG construction above was formulated for a single property. However, a given trace might generate many potential warnings for concurrency bugs and building the UCG from scratch for each warning would be infeasible in practice. In order to build a single UCG for all the warnings, we start by adding the seed edges for all the warnings. The seed edges for a given property are now labeled with an id that is unique to that property. If the same seed edge needs to be added for multiple properties then it is labeled with the set of ids of all these properties. During the UCG construction

via alg. 1 when an edge e induces another edge f, then the property-ids are propagated from e to f by relabeling f with the union of ids of e and f's pre-existing ids. In this way each edge in the UCG may be labeled with multiple property-ids. It is easy to see that an edge will occur in the UCG of a property P if and only if it is labeled with P's id (and possibly other ids).

The resulting UCG U contains edges that are induced by all the properties. If we are interested in checking whether a given property P holds, then we can extract from U the UCG induced by P by simply projecting on to the edges that are labeled with P's id. Then P is satisfied if and only if the resulting sub-graph is acyclic.

The above technique of propagating the ids of properties starting from the seed edges allows us to build the UCG only once while the validity of the different properties can be checked separately by projecting onto the appropriate sub-graphs. Note that since the wait/notify seed edges occur in the UCGs for all the properties, they are labeled with ids of all the properties.

4 Chopping Result: Scaling UCG Construction

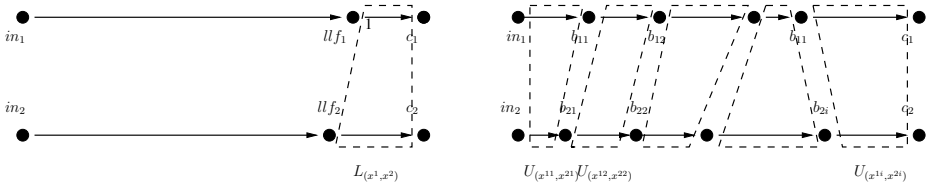
In order to leverage the UCG for a practically feasible analysis we have to address the key issue that the number of constraints added to the UCG may be too large. This is because (1) the traces x^1 and x^2 could be arbitrarily long, and (2) wait/notify events could be many and could span the entire lengths of these traces. Thus a very large number of wait/notify seed edges, and, as a result, induced edges, could be added along the entire lengths of x^1 and x^2. It contrast (see fig. 3), when constructing the lock causality graph (LCG) as in [10] for reasoning about threads interacting only via locks, causality edges are added only between lock/unlock statements occurring along the suffixes of x^1 and x^2 starting at their last lock-free states. In practice, these suffixes of x^1 and x^2 are short, as for performance reasons programmers tend to keep the lengths of critical sections small. This ensures that the LCG size is small thereby ensuring scalability.

Decomposition Result. In order to guarantee scalability of the UCG construction in the presence of both wait/notifies and locks, our goal, analogous to LCGs in [10], is to restrict the UCG construction to only small suffixes of $x^1, ..., x^n$. Towards that end, we start with the following key *decomposition result* which provides useful insight into the structure of UCGs. Intuitively, the decomposition result states that the given paths $x^1, ..., x^n$ can be broken down into an equal number, say m, of segments, with $x^j = x^{j1}...x^{jm}$ such that in order to check the acyclicity of $U_{(x^1,...,x^n)}(P)$ it suffices to check the acyclicity of each of the m smaller UCGs $U_{(x^{1i},...,x^{ni})}(P)$.

Theorem 2. (Partitioning Result). *Given finite local computations $x^1, ..., x^n$ of $T_1, ..., T_n$ respectively, for each j, let $x^j = x^{j1}x^{j2}$ be a partition of x^j such that*

- *the last state occurring along x^{j1} is lock-free, and*
- *for $j \neq k$, there does not exist a wait/notify seed edge, a fork-join seed edge or a property seed edge with endpoints along x^{j1} and x^{k2} or along x^{k1} and x^{j2}.*

Then $U_{(x^1,...,x^n)}(P)$ is acyclic if and only if $U_{(x^{11},...,x^{n1})}(P)$ and $U_{(x^{12},...,x^{n2})}(P)$ are acyclic.



(a) Lock Causality Graph (b) Universal Causality Graph

Fig. 3. Universal Causality Graph Decomposition

Repeated application of the above result leads to the following partitioning result.

Corollary 3. (Decomposition Result). *Given finite local computations $x^1, ..., x^n$ of threads $T_1, ..., T_n$, respectively, for each j, let $x^j = x^{j1}...x^{jm}$ be a partition of x^j such that*

- *the last states occurring along x^{ji}, where $i \in [1..m]$, are lock-free, and*
- *there does not exist a wait/notify seed edge, a property seed edge or a fork-join seed edge with endpoints in x^{ji} and $x^{ki'}$, where $j \neq k$ and $i \neq i'$.*

Then $U_{(x^1,...,x^n)}(P)$ is acyclic if and only if for each $i \in [1..m]$, $U_{(x^{1i},...,x^{ni})(P)}$ is acyclic.

The situation is illustrated in fig. 3. The lock causality graph, shown in 3(a), is generated only by the suffixes of x^1 and x^2 starting with the last lock free states llf_1 and llf_2 along x^1 and x^2, respectively. However the UCG for x^1 and x^2 is comprised of the UCGs $U_{(x^{1i},x^{2i})}(P)$ (and some more edges which don't impact its acyclicity) generated by the pairs of segments x^{1i} and x^{2i} delineated, respectively, by the *causality barriers* b_{1i} and $b_{1(i+1)}$, and b_{2i} and $b_{2(i+1)}$, where a causality barrier is as defined below:

Definition (Causality Barrier). *Given finite local computations $x^1, ..., x^n$ of threads $T_1, ..., T_n$, respectively, where $x^i = x^i_0...x^i_{n_i}$, we say that the n-tuple $(x^1_{b_1}, ..., x^n_{b_n})$, with $x^i_{b_i}$ being a local state of T_i, forms a causality barrier if (1) for each i, $x^i_{b_i}$ is lock-free, i.e., no lock is held by T_i at $x^i_{b_i}$, and (2) there does not exist a seed edge $(x^j_m, x^k_{m'})$, where $j \neq k$, $m \in [0..b_j]$ and $m' \in [b_k + 1, n_k]$ or $m \in [0..b_k]$ and $m' \in [b_j + 1, n_j]$.*

Intuitively, seed edges along the traces x^1 and x^2 gives rise to *localized* universal causality graphs that are separated by causality barriers.

Chopping Result for Predictive Analysis. In predictive analysis, we start from a global execution trace x of the program. Our goal is to decide whether there exists a different valid interleaving of the local computations x^1 and x^2 of T_1 and T_2 along x, that may uncover an error. If we were given two arbitrary local computations y^1 and y^2 of threads T_1 and T_2 then in order to decide whether there exists an interleaving of y^1 and y^2 leading to an error state, we would have to build the complete UCG along the entire lengths of y^1 and y^2. However, by exploiting the fact in predictive analysis, x^is are projections of a valid global computation x onto the local states of individual threads, we now show that we need not build the entire UCG $U_{(x^1,x^2)}(P)$ but only the one generated by suffixes x^{1b} and x^{2b} of x^1 and x^2, respectively, starting at a last barrier pair along x^1 and

Algorithm 2. Computing a Last Causality Barrier

1: **Input:** A pair of local paths x^1 and x^2 leading to local states c_1 and c_2 of threads T_1 and T_2, respectively.
2: Let lf_1 be the last lock-free state before c_1 along x^1 such all (start or end) vertices of property edges occur after lf_1 along x^1 and let WN_1 be the set of wait/notify events encountered along the segment $x^1_{[lf_1,c_1]}$, i.e., between local states lf_1 and c_1 along x^1
3: Set *terminate* to *false* and lf_2 to c_2
4: **while** *terminate* equals *false* **do**
5: Let lf'_2 be the last lock-free state before lf_2 along x^2 such that each wait/notify event in WN_1 is matched by an event along the segment $x^2_{[lf'_2,c_2]}$ and all (start or end) vertices of property edges occur after lf_2 along x^2. Let WN_2 be the set of wait/notify events encountered along $x^2_{[lf'_2,lf_2]}$
6: Let lf'_1 be the last lock-free state at or before lf_1 along x^1 such that each wait/notify event in WN_2 is matched by an event along the segment $x^1_{[lf'_1,c_1]}$. Let WN'_1 be the set of wait/notify events encountered along $x^1_{[lf'_1,lf_1]}$
7: **if** lf'_1 equals lf_1 **then**
8: Set $terminated = true$ and **output** (lf_1, lf'_2) as a last causality barrier
9: **else**
10: Set $WN_1 = WN'_1$, $lf_1 = lf'_1$ and $lf_2 = lf'_2$
11: **end if**
12: **end while**

x^2. This ensures scalability of our analysis as we can, in practice, ignore most synchronization primitives except for the last few. We say that the n-tuple $(x^1_{b_1}, ..., x^n_{b_n})$ of local states of threads $T_1, ..., T_n$ is a *last causality barrier* along $x^1, ..., x^n$ if there does not exist another causality barrier $(x^1_{b'_1}, ..., x^n_{b'_n})$ such that for each i, $x^i_{b'_i}$ occurs after $x^i_{b_i}$ along x^i and all property seed edges are of the form $a \rightsquigarrow b$, where, for some i, j, a and b occur after $x^i_{b_i}$ and $x^j_{b_j}$ along x^i and x^j, respectively. Then

Theorem 4. (Chopping Result). *Let $x^1, ..., x^n$ be local computations of threads $T_1, ..., T_n$, respectively, along a valid global computation x of the given concurrent program. Let $U_{(x^{1b},...,x^{nb})}(P)$ be the UCG generated by the suffixes $x^{1b}, ..., x^{nb}$ of $x^1, ..., x^n$, respectively, beginning with a last causality barrier $(x^1_{b_1}, ..., x^n_{b_n})$ along $x^1, ..., x^n$. Then property P is violated via a statically feasible interleaving of $x^1, ..., x^n$ if and only if $U_{(x^{1b},...,x^{nb})}$ is acyclic.*

Computing a Last Casualty Barrier. As the final step, we present a procedure (alg. 2) to identify a last causality barrier. For simplicity, we handle only the two thread case. Let c_1 and c_2 be the last local states along x^1 and x^2, respectively. From c_1 we traverse backwards along x^1 till we reach the last lock free state lf_1 along x^1 before c_1. Note that all the wait/notify events occurring between lf_1 and c_1 along x^1, denoted by WN_1, must be matched along the suffix beginning with $x^2_{b_2}$. Therefore from c_2, we have to traverse backward till we encounter the first lock-free state lf_2 such that all events in WN_1 are matched along the suffix of x^2 starting at lf_2. However, in traversing backward from c_2 to lf_2 we may encounter wait/notify events, denoted by the set WN_2, that are not matched along the suffix of x^1 starting at lf_1. In that case, we need to traverse further

backwards starting at lf_1 till we encounter a lock-free state lf'_1 such that all events in WN_2 are matched along the suffix of x^1 starting at lf'_1. If we do not encounter any new wait/notify event that is unmatched along the suffix of x^2 starting at lf_2 then we have reached a fixpoint. Else if there exist wait/notify events occurring along the suffix of x^1 starting at lf'_1 that are unmatched along the suffix of x^2 starting at lf_2 then the whole procedure is repeated till a fixpoint is reached.

5 Experiments

We have implemented the proposed algorithm in a tool called *Fusion* [17]. Our tool is capable of analyzing execution traces generated by both Java programs and multi-threaded C programs using *PThreads*. For C programs, we use CIL [13] to instrument the source code to create executables that can log execution traces at runtime. For Java programs, we use execution traces logged at runtime by a modified version of the Java PathFinder. The Java traces used in our experiments were kindly provided to us by Mahmoud Said. Our experiments were conducted on a PC with 1.6 GHz Intel processor and 2GB memory running Fedora 8.

The overall predictive analysis is as follows. We first find the potential errors (warnings) by a simple static analysis; these are event pairs for data races and event triplets for atomicity violations. Then we apply the UCG analysis to prune away as many spurious warnings as we can. Finally, we use a SMT-based procedure (as in [17]) to check the remaining UCG warnings. This final step uses the *Yices* SMT solver [3]. For each reported error, the SMT-based procedure also returns a witness trace. The UCG warnings are checked one by one in the SMT-based procedure, but we use the incremental feature of the SMT solver to share the cost of checking different warnings. We also add the induced constraints of the UCG to the SAT solver to help speed up the search.

Table 1 shows the results of predicting data races and three-access atomicity violations [17] in traces of Java and C programs. All benchmarks are public domain[1]. The first two columns show the name and the number of threads. The next five columns show the statistics of the trace, including the number of events, the number of lock events, the number of wait-notify events, the number of lock variables, and the number of condition variables. The next six columns show the results of predicting data races using both static analysis and model checking. The first four columns are the total number of warnings, the number of warnings after a lock based analysis alone (lsa), the number of warnings after a fork/join/wait/notify analysis alone (mhb), and the number of warnings after the combined UCG analysis (ucg). The next two columns shows the results of model checking the UCG warnings, including the number of witnesses generated and the model checking time in seconds.

The last six columns of Table 1 show the results of predicting three-access atomicity violations. The data format is the same as predicting data races, except that the warnings are now potential atomicity violations. Note that in order to predict atomicity violations, the user transactions (which are intended to be atomic) need to be marked explicitly in the traces. For Java traces, we have assumed that all the synchronized blocks are

[1] The traces are available at
http://www.nec-labs.com/$\sim$chaowang/pubDOC/LnW.tar.gz

Table 1. Predicting data races and atomicity violations in traces of Java/C programs

Test Program		Given Trace					Predicting Data Races						Predicting Atomicity Violations					
		# events			# vars		static ana. (warnings)				witness gen		static ana. (warnings)				witness gen	
name	thrd	total	lk	wn	lk	wn	total	lsa	mhb	ucg	wits	time(s)	total	lsa	mhb	ucg	wits	time(s)
ex.race	3	29	4	0	2	0	8	8	2	**2**	1	0.1	2	2	0	**0**	0	0.0
ex.norace	3	37	8	0	2	0	8	6	2	**0**	0	0.0	2	2	0	**0**	0	0.0
ra.Main	3	55	12	5	3	4	13	13	4	**4**	1	0.1	2	2	0	**0**	0	0.0
connectionpool	4	97	16	5	1	3	89	21	28	**0**	0	0.0	30	6	4	**0**	0	0.0
liveness.BugG	7	285	39	6	9	6	408	138	280	**10**	0	0.4	280	60	220	**0**	0	0.0
s1.JGFBarrier	10	649	62	21	2	7	1831	488	1214	**30**	0	1.4	852	102	612	**3**	0	1.6
s1.JGFBarrier	13	799	77	28	2	7	2952	656	2077	**49**	0	3.6	950	87	709	**9**	0	3.7
account.Main	11	902	146	12	21	10	372	342	186	**162**	20	4.0	140	140	60	**60**	2	1.3
philo.Philo	6	1141	126	41	6	22	1719	566	576	**0**	0	0.0	413	81	177	**0**	0	0.0
s1.JGFSyncB	16	1510	237	0	2	0	21230	1142	17415	**117**	0	800	13578	186	11532	**0**	0	0.0
account.Main	21	1747	282	20	41	20	740	680	420	**360**	80	54.5	280	280	120	**120**	3	5.9
elevator.E	4	3000	368	0	11	0	1293	1276	17	**0**	0	0.0	6	4	2	**0**	0	0.0
elevator.E	4	4998	587	0	11	0	3178	3128	50	**0**	0	0.0	12	8	4	**0**	0	0.0
elevator.E	4	8000	1126	0	11	0	3553	3458	95	**0**	0	0.0	18	12	6	**0**	0	0.0
tsp.Tsp	4	45653	20	5	5	3	113	113	4	**4**	3	0.1	0	0	0	**0**	0	0.0
atom001	3	88	6	0	1	0	8	5	3	**0**	0	0.0	4	4	2	**2**	1	0.1
atom001a	3	100	8	1	1	1	12	8	4	**0**	0	0.0	4	4	2	**2**	0	0.1
atom002	3	462	124	0	2	0	96	45	51	**0**	0	0.0	68	68	34	**34**	33	36.7
atom002a	3	462	126	3	2	1	101	49	52	**0**	0	0.0	68	68	34	**34**	0	32.0
banking-av	3	748	20	0	3	0	284	284	72	**72**	72	3.1	64	64	32	**32**	32	1.5
banking-sav	3	852	28	2	3	2	333	325	80	**72**	24	5.2	64	64	32	**32**	16	3.4
banking-noav2	3	856	32	2	3	2	337	305	80	**48**	0	1.3	64	48	32	**16**	0	1.2

intended to be atomic, unless the synchronized block has a *wait* (in which case it is clearly intended to be non-atomic). For the C programs used in this experiment, we have manually annotated certain blocks in the program source code as intended-to-be-atomic. Note that in all the examples the runtime of the UCG-based analysis is negligible in comparison to the model checking time.

The results in Table 1 show that, if one relies on either the lock analysis alone or the fork-join-wait-notify based analysis alone, the number of (spurious) warnings (for data races or atomicity violations) can be large. In contrast, our UCG based analysis, by exploiting the interaction among these two types of happens-before constraints, is effective in pruning away spurious warnings. Also note that, even with the significantly improved precision, the number of UCG warnings can still be large, e.g. for human beings to inspect manually. Therefore, in our predictive analysis framework a precise SMT-based algorithm [16, 17] is used at the final step to check all the UCG warnings. The algorithm is precise in that it generates witness traces if and only if the UCG warnings are indeed real errors. In the end, all the witnesses generated can be fed to a special thread scheduler in the *Fusion* tool, to re-run the program and deterministically replay the actual violation.

6 Conclusions

We have proposed the notion of a *Universal Causality Graph (UCG)*, as a unified happens-before model for detecting bugs in concurrent programs. Given a concurrent (trace) program and a property, UCGs allow us to capture, as causality constraints, the

set of all possible interleavings that are feasible under the scheduling constraints imposed by synchronization primitives and that may potentially lead to violations of the property while guaranteeing (i) soundness and completeness, (ii) scalability, (iii) handling of multiple synchronization primitives in a unified manner, and (iv) exploiting the synergy between causal constraints imposed by the property as well as the program. As an application, we demonstrated the use of UCGs in enhancing the precision and scalability of predictive analysis in the context of runtime verification of concurrent programs.

References

[1] http://www.cs.utexas.edu/users/kahlon/

[2] Chen, F., Rosu, G.: Parametric and sliced causality. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 240–253. Springer, Heidelberg (2007)

[3] 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)

[4] Elmas, T., Qadeer, S., Tasiran, S.: Goldilocks: a race and transaction-aware java runtime. In: PLDI (2007)

[5] Farzan, A., Madhusudan, P.: Monitoring atomicity in concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 52–65. Springer, Heidelberg (2008)

[6] Farzan, A., Madhusudan, P.: The complexity of predicting atomicity violations. In: TACAS (2009)

[7] Farzan, A., Madhusudan, P., Sorrentino, F.: Meta-analysis for atomicity violations under nested locking. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 248–262. Springer, Heidelberg (2009)

[8] Flanagan, C., Freund, S.N.: Atomizer: A dynamic atomicity checker for multithreaded programs. In: IPDPS (2004)

[9] Flanagan, C., Freund, S.N., Yi, J.: Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. In: PLDI (2008)

[10] Kahlon, V.: Boundedness vs. Unboundedness of Lock Chains: Characterizing Decidability of CFL-Reachability for Threads Communicating via Locks. In: LICS (2009)

[11] Kahlon, V., Ivancic, 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)

[12] Lu, S., Tucek, J., Qin, F., Zhou, Y.: AVIO: detecting atomicity violations via access interleaving invariants. In: ASPLOS (2006)

[13] Necula, G., McPeak, S., Rahul, S., Weimer, W.: CIL: Intermediate language and tools for analysis and transformation of c programs. In: CCC (2002)

[14] Sadowski, C., Freund, S.N., Flanagan, C.: Singletrack: A dynamic determinism checker for multithreaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 394–409. Springer, Heidelberg (2009)

[15] Sen, K., Rosu, G., Agha, G.: Detecting errors in multithreaded programs by generalized predictive analysis of executions. In: Steffen, M., Zavattaro, G. (eds.) FMOODS 2005. LNCS, vol. 3535, pp. 211–226. Springer, Heidelberg (2005)

[16] Wang, C., Kundu, S., Ganai, M., Gupta, A.: Symbolic predictive analysis for concurrent programs. In: ISFM (2009)

[17] Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: TACAS (2010)

[18] Wang, L., Stoller, S.D.: Runtime analysis of atomicity for multithreaded programs. IEEE Trans. Software Eng. 32(2), 93–110 (2006)

Automatically Proving Linearizability

Viktor Vafeiadis

University of Cambridge

Abstract. This paper presents a practical automatic verification procedure for proving linearizability (i.e., atomicity and functional correctness) of concurrent data structure implementations. The procedure employs a novel instrumentation to verify logically pure executions, and is evaluated on a number of standard concurrent stack, queue and set algorithms.

1 Introduction

Linearizability [11] is the standard correctness requirement for concurrent implementations of abstract data structures (such as stacks, queues, sets and finite maps) packaged into a concurrent library (such as `java.util.concurrent`). It requires every library operation to be atomic (behave as if it were executed in one indivisible step) and to satisfy a given functional correctness specification.

The most common way to prove linearizability is to identify the so-called *linearization points* of each operation. These are program points where the entire effect of an operation execution logically takes place. Sadly, however, these linearization points are often rather complicated: they can depend on a non-local boolean condition and can even reside within other concurrently executing threads. This makes a brute force search for the linearization points infeasible.

We observe, however, that in practice such complicated linearization points arise only in operation executions that do not *logically* update the library's shared state. It is therefore possible to search for the linearization points for operations whose specification is always effectful (i.e., modifies the shared state), but we need a different approach to verify operations with a possibly pure specification (i.e., one not modifying the shared state).

This paper presents one such procedure for proving linearizability (see §4). For operations with a possibly pure specification, it instruments the library code with a certain 'pure linearizability checker,' derived from the specification, and runs a suitably powerful abstract interpreter to validate that there are no assertion violations. This effectively considers all possible linearization points in one go and results in a non-constructive linearizability proof. As a result, we have succeeded in verifying several concurrent stack and queue implementations, and have obtained mixed results for set implementations (see §5 for details).

Related Work. The related verification work can be classified in three groups.

First, there are various model-checking papers [2,2,13,4]. These do not prove correctness; they merely check short execution traces of a small number of

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 450–464, 2010.

```
Sequence AQ = @empty;

void enqueue(int e) {
  atomic {
    AQ = @app(AQ, @singl(e));
  }
}
```

```
int tryDequeue(void) { int ARes;
  atomic {
    if (AQ == @empty) return EMPTY;
    else { ARes = @hd(AQ);
           AQ = @tl(AQ);
           return ARes; }
} }
```

Fig. 1. Specification of a concurrent queue object

threads. On the positive side, such tools do not require linearization points to be annotated, are good at quickly finding bugs, and return concrete counterexample traces for failed verifications.

Second, there are static analyses (shape analyses, in particular) [2,3,19]. With the exception of [2], these analyses work for an unbounded number of threads and result in a proof of linearizability. Unfortunately, all of these analyses require the programmer to specify the linearization points, a task that is quite difficult when the linearization points are conditional or within the source code of other concurrently executing operations, as we will shortly see. Our paper addresses this limitation of the existing static analyses.

Finally, there are manual verification efforts. Some (e.g.,[18]) are pencil and paper proofs in a particular program logic, others (e.g.,[5]) do a direct simulation proof in a mechanised proof assistant, while O'Hearn et al. [15] do part of the proof in a program logic and another part using operational reasoning on traces.

2 A Motivating Example: The M&S Queue

We start with a motivating example for the rest of the paper: the well-known Michael & Scott non-blocking queue [14] (henceforth referred to as the M&S queue). Figure 1 contains the specification of the concurrent queue operations written in C-like pseudocode. The state of the queue is represented by the shared variable `AQ`, which holds a sequence of values. We use the following notation for mathematical sequences: `@empty` stands for the empty sequence; `@singl` constructs a sequence consisting of one element; `@app` concatenates two sequences; `@hd` returns the first element of a sequence; and `@tl` returns all but the first element of a sequence.

The queue supports two operations: (*i*) `enqueue`, which adds an item to the end of the queue, and (*ii*) `tryDequeue`, which removes and returns the first item of the queue if there is one, or returns `EMPTY`, if the queue is empty. Both operations are supposed to be *atomic*; that is, executing in one step.

Figure 2 contains the M&S queue implementation, which is significantly more complicated than the specification above. The queue is represented by two pointers into a null-terminated singly-linked list. The first pointer (`Q->head`) points to the beginning of the list and is updated by `tryDequeue` operations. The second pointer (`Q->tail`) is used to find the end of the list so that `enqueue` can

```
typedef struct Node_s *Node;

struct Node_s {
  int val;
  Node tl;
}

struct Queue {
  Node head, tail;
} *Q;

void enqueue(int value) {
  Node node, next, tail;
  node = new_node();
  node->val = value;
  node->tl = NULL;
  while(true) {
    tail = Q->tail;
    next = tail->tl;
    if (Q->tail != tail) continue;
    if (next == NULL) {
      if (CAS(&tail->tl,next,node))
        break;
    } else {
      CAS(&Q->tail,tail,next);
    }
  }
  CAS(&Q->tail,tail,node);
}
```

```
void init(void) {
  Node node = new_node();
  node->tl = NULL;
  Q = new_queue();
  Q->head = node;
  Q->tail = node;
}

int tryDequeue(void) {
  Node next, head, tail;
  int pval;
  while(true) {
    head = Q->head;
    tail = Q->tail;
    next = head->tl;
    if (Q->head != head) continue;
    if (head == tail) {
      if (next == NULL)
        return EMPTY;
      CAS(&Q->tail,tail,next);
    } else {
      pval = next->val;
      if (CAS(&Q->head,head,next))
        return pval;
    }
  }
}
```

Fig. 2. The M&S non–blocking queue implementation

locate the last node of the list. It does not necessarily point to the last node of the list, but it can lag behind. This is because there is no widely available hardware instruction that can change `Q->tail` and append one node onto the list in one atomic step. Consequently, `enqueue` first appends a node onto the list with the underlined `CAS` instruction, and later updates `Q->tail` with its final `CAS` instruction. In addition, whenever a concurrently executing thread notices that the tail pointer is lagging behind the end of the list, it tries to advance it using the `CAS(&Q->tail,tail,next)` instructions.

In the remainder of this paper we shall define what it means for the implementation to satisfy its specification and present a method for proving this.

3 Linearizability

We take programs to consist of a sequential initialisation phase followed by a parallel composition of a fixed (but not statically bounded) number of threads, T.

The state consists of a set of global variables, G, and a set of local variables per thread, L_t, that includes the thread's program counter, $\mathtt{pc}_t$. As a convention, we will subscript thread-local variables with the corresponding thread identifier to distinguish them from the global variables. We model each thread as a transition relation on the valuations of the global and its local variables.

A library, A, consists of a constructor, A_{init}, and a number of operations (a.k.a., methods), $A_1, \ldots, A_n$, each expecting a single argument, $\mathtt{arg}_t$, and returning their result in the thread-local variable $\mathtt{res}_t$. A client of the library is any program that calls the library's constructor once in its initial sequential phase, and then can call any number of the library's methods possibly concurrently with one another. Let $C[A]$ be the transition relation denoting the composition of the client C with the library A. We write $C[A]^*$ for its reflexive and transitive closure. In such a composition, we write G^C (resp. L_t^C) for the global (resp. local) variables belonging to the client and, analogously, G^A and L_t^A for those belonging to the library. We assume that G^C and G^A are disjoint, and that $L_t^C \cap L_t^A = \{\mathtt{arg}_t, \mathtt{res}_t, \mathtt{pc}_t\}$.

Linearizability [11] is a formalisation of the concept of atomicity. Briefly, it requires that every execution history consisting of calls to `enqueue` and `tryDequeue` is equivalent (up to reordering of events) to a legal, sequential history that preserves the order of non-overlapping methods in the original history. We say that a history is sequential if none of its methods overlap in time; moreover, it is legal if each method satisfies its specification.

In this paper, we prefer a slightly different definition of linearizability given in terms of *instrumented* clients.

Definition 1. *An* instrumented *client of a library A is a client annotated with an auxiliary global variable h as follows: (1) At the initial state, let $h = \epsilon$; (2) before every call to A_i by thread t, append* $(\textsc{call}\ t, i, \mathtt{arg}_t)$ *to h; and (3) after each return from a call to A_i by thread t, append* $(\textsc{ret}\ t, i, \mathtt{res}_t)$ *to h.*

In effect, the auxiliary variable h records the observed execution history. Note that there is a gap in time between when a method returns and when the return is recorded in h. This gap allows us to define linearizability as follows:

Definition 2 (Linearizability). *A library A is* linearizable *with respect to a specification B if and only if for all instrumented clients C and every state s, if $(s_{\text{init}}, s) \in C[A]^*$, then there exists a state s' such that $(s'_{\text{init}}, s') \in C[B]^*$ and $s(h) = s'(h)$, where s_{init} and s'_{init} are the initial states after calling A_{init} and B_{init} respectively.*

This definition is slightly easier to work with than the original one by Herlihy and Wing [11], because it uses syntactic equality on the recorded histories rather than equivalence up to reordering of non-overlapping calls of the actual histories.

It is also more general in the sense that it corresponds to the original definition only if all of B's methods are atomic. The same generalisation is found in the definition of Filipović et al. [7].

3.1 Proving Linearizability Using Linearization Points

The most common way of proving linearizability of a concurrent library is to find the so-called linearization points of each operation and to demonstrate that the chosen points are correct. These are points in the source code of the library which, when executed, are deemed to perform the entire observable effect of the operation instantaneously, and hence define the order in which the concurrent operations are to be linearized. Within each operation execution, exactly one linearization point must occur, but statically there can be multiple such points along different execution paths of the operation, some of which might not even be inside the operation's source code!

Linearization Points of the M&S Queue. The linearization point of `enqueue` is the underlined `CAS` instruction, when it succeeds. Its effect is to link a node to the end of the concrete list, which logically corresponds to appending an item to the queue.

The `tryDequeue` method has two linearization points depending on the result. The linearization point for the empty case is the underlined assignment `next = head->tl`. This is a linearization point only if the same loop iteration later executes `return EMPTY`. The second linearization point is the underlined `CAS` instruction. Its effect is to advance the `Q->head` pointer, which logically removes the first element from the queue.

As presented, these linearization points are conditional: not every time the underlined instructions are executed, they are linearization points. Fortunately, the conditions of the two points involving `CAS` can easily be eliminated by unfolding the definition of `CAS`. For example, if we expand out the definition of the underlined `CAS` of `enqueue`, we get:

```
atomic { if (tail->tl == next) { tail->tl = node; break; } }
```

Thus, it is easy to identify the linearization point of `enqueue` with the underlined assignment to `tail->tl` whenever that assignment is executed. We can do likewise with the second linearization point of `tryDequeue`.

In contrast, the first linearization point of `tryDequeue` is truly conditional. Specifying it formally requires an auxiliary prophecy variable [1] to record whether the program will later execute `return EMPTY` in the same loop iteration. The prophecy variable is needed because when executing the underlined read from `head->tl` we cannot tell whether the test `Q->head != head` on the following line will succeed. Therefore, the full condition is:

$$\neg\texttt{prophecy(Q->head!=head)} \land \texttt{head==tail} \land \texttt{next==NULL}.$$

In §4.2, we will see a technique for proving that `tryDequeue` is linearizable that avoids the conditions on this linearization point and the prophecy variable.

4 Automatic Proof Technique

4.1 Key Observation

It is clear from the M&S queue that linearization points are often conditional, and that some conditions can be quite involved. Searching for such complex conditions is clearly infeasible. We can, however, observe that

> Operations have complex conditional linearization points
> only in executions that do not *logically* modify the state.

For example, at the first linearization point of `tryDequeue`, the operation does not logically modify the state. That is, if `AQ` holds the logical contents of the queue and we execute the `tryDequeue` specification at that point, the value of `AQ` will not be affected. It is, however, possible that `tryDequeue` updates the concrete state (e.g., by performing the `CAS(&Q->tail, tail, next)` in a previous loop iteration), but these updates do not affect its logical contents.

Quite surprisingly, this observation holds for most concurrent algorithms in the literature. To the best of our knowledge, it holds for all but two of the algorithms in Herlihy & Shavit's book [12]. A possible explanation as to why this is so is that logically effectful operation executions are much more difficult to optimise than the ones that only do not logically modify the state. Therefore, they tend to have simpler correctness arguments than the more heavily optimised logically pure executions. Notable exceptions where our observation does not hold are: (*i*) the Herlihy & Wing queue [11], (*ii*) the elimination-based stack of Hendler et al. [10], and (*iii*) RDCSS by Harris et al. [9]. Verifying these algorithms automatically is beyond the scope of this paper.

In the following, we shall distinguish between *pure* and *effectful* executions of the abstract operations, i.e. the operation specifications. We say that an abstract operation execution is pure if it does not modify the abstract state. Otherwise, we say that the execution is effectful.

4.2 Dealing with Logically Pure Executions

To verify logically pure executions, we introduce one auxiliary boolean array, $\texttt{can\_return}_{t,op}[]$, per thread and per library operation. Each array is indexed by the set of possible return values. While thread t is executing the operation op, then $\texttt{can\_return}_{t,op}$ satisfies the following invariant: whenever an entry, $\texttt{can\_return}_{t,op}[v]$, in the array is true, then there exists an instant since the operation was called at which if the operation's specification had been executed, it would not have modified the global (abstract) state and would have returned v. Therefore, if $\texttt{can\_return}_{t,op}[\texttt{res}_t]$ is true when the operation returns, we know that there existed a valid linearization point during the operation's execution.

To ensure that the aforementioned invariant holds, we set all the elements of `can_return[]` to false at the beginning of the operation. Then, at any later point, we can set `can_return`$[v]$ to true provided that executing the operation's specification does not modify the global (abstract) state and returns v. This is the task of the 'pure linearizability checker,' which we introduce below.

Pure Linearizability Checker Construction. Assuming that the specifications do not contain any loops or any function calls, we rewrite each specification as a non-deterministic choice of a number of execution paths consisting of assignments, assume statements, and terminated by a return command. For uniformity, we change specifications that do not return any value to return 0. For example, the `enqueue` and `tryDequeue` specifications become:

$$\begin{aligned} enq &\stackrel{\text{def}}{=} \texttt{AQ=@app(AQ,@singl(e)); return 0} \\ deq(1) &\stackrel{\text{def}}{=} \texttt{assume(AQ==@empty); return EMPTY} \\ deq(2) &\stackrel{\text{def}}{=} \texttt{assume(AQ}\neq\texttt{@empty); ARes=@hd(AQ); AQ=@tl(AQ); return ARes} \end{aligned}$$

where `tryDequeue` corresponds to the non-deterministic choice among the paths $deq(1)$ and $deq(2)$. We say that a path is *syntactically pure* if and only if it has no assignments to global variables. For example, $deq(1)$ is syntactically pure, but enq and $deq(2)$ are not.

The pure linearizability checker is constructed by replacing `return` v with `can_return`$[v]$`=true` along every syntactically pure specification path of the method, and by truncating the non-pure paths before their first effectful command (namely, an assignment to a global variable). This construction ensures that pure linearizability checkers set `can_return`$[v]$ to true *only if* at the current point the specification does not modify the global state and returns v.

Going back to the queue specifications, the pure linearizability checker of `enqueue` is simply the empty command, because enq is not syntactically pure. The pure linearizability checker of `tryDequeue` is

```
if(*) {assume(AQ==@empty); can_return[EMPTY]=true;}
else  {assume(AQ!=@empty); ARes=@hd(AQ);}
```

In this case, as `ARes` is a dead local variable, the assignment can be removed, and the checker can be rewritten as follows:[1]

```
if(AQ==@empty) {can_return[EMPTY]=true;}
```

Linearization Points in Other Threads. Note that it is sound to execute the pure checker for a thread, t, at any point in time, even between atomic steps of other threads. This allows us to handle linearization points that are in the source code of other concurrently executing operations. Basically, when a verifier checks one thread with a compositional verification technique, it has a model of what updates all the other threads can do and how these updates affect the current thread. Thus, when symbolically evaluating the operation being verified, after each of its atomic commands, the static analyser also symbolically evaluates the model of what all the other threads can do, before proceeding with the operation's next atomic command. It suffices, therefore, to adapt the verifier to call the relevant pure linearizability checker is also called after each atomic step

[1] This simplification is for presentation purposes only. Our implementation does not perform such simplifications.

Algorithm 1. ProveLinearizable($op_{\text{init}}, spec_{\text{init}}, op_1, spec_1, \ldots, op_n, spec_n$)

```
1: iop_init ← (op_init; spec_init)
2: for i ← 1 to n do
3:     check_i ← GeneratePureChecker(spec_i)
4: (C, op_1, ..., op_n) ← GetCandidateLinPoints(op_1, ..., op_n)
5: for all cand ∈ C do
6:     for i ← 1 to n do
7:         iop_i ← InstrumentLinPoints(cand, op_i, spec_i)
8:     if Verify(iop_init, iop_1, check_1, ..., iop_n, check_n) then
9:         return 'Success'
10: return 'Failure'
```

of its model of the orther threads. This enables us to establish linearizability even in cases where some linearization points are within a different thread. The exact details as to how this is implemented are in §4.5.

4.3 Verification Procedure Outline

Algorithm 1 contains our procedure for proving linearizability. ProveLinearizable takes as arguments the library's constructor (op_{init}) with its specification ($spec_{\text{init}}$), and the set of library operations ($op_1, \ldots, op_n$) with their specifications ($spec_1, \ldots, spec_n$). The specifications are just normal methods that operate on the logical state, which is disjoint from the concrete state.

The algorithm consists of two phases. First, it instruments the constructor of the library, computes the pure checkers for each operation and generates a set of candidate linearization point assignments, $\mathcal{C}$. Then, it iterates over that set checking whether any of these assignments is valid. If a valid assignment is found, the procedure returns 'Success' indicating that linearizability has been proved; otherwise, it returns 'Failure.'

Preparation Phase. First, the algorithm instruments the library's constructor: iop_{init} is simply the sequential composition of the constructor, op_{init}, and its specification, $spec_{\text{spec}}$. Next, pure checkers are generated, as described in §4.2.

Then, GetCandidateLinPoints is called. This, first, unfolds the definitions of `CAS` and `DCAS` in the various operations. This syntactic transformation exposes the trivial conditions governing the linearization points of effectful operations, so that the transformed operation has unconditional linearization points. For uniformity, it arranges that methods and specifications that do not return any results, return 0 instead.

Then, along each execution path of each operation, it chooses one command writing to the shared state as the effectful linearization point. If the operation's specification has pure executions (e.g., `tryDequeue`), it also can choose no linearization point on some of its execution paths in the hope that the execution path corresponds to pure execution of its specification. Obviously, memory writes appearing within loops are discarded since they can be executed multiple times. This process produces one linearization point assignment: a set of

program points that are to be treated as (unconditional) linearization points of the method they belong to. GetCandidateLinPoints returns the set, $\mathcal{C}$, of all possible linearization point assignments.

Checking Phase. Each operation op_i is instrumented with its specification $spec_i$ by adding the two new auxiliary local variables:

- `lres`, holding the result of the abstract method call at the effectful linearization point if this has occurred, or the reserved value `UNDEF` otherwise,
- `can_return`, an array storing the allowed return values of any pure linearization points that have been executed so far,

and the following code:

- At the beginning of the method, InstrumentLinPoints sets `lres` to `UNDEF`. and all the elements of `can_return`[] to false.
- At the chosen candidate linearization points in *cand* that are in the source code of op_i, it inserts an assertion checking that the linearization point has not occurred followed by a call to the abstract method:

  $$\texttt{assert(lres==UNDEF); lres=}spec_i\texttt{(args)}$$

  where `args` are the arguments of op_i (which we assume are not modified by op_i). The assertion about `lres` and the subsequent assignment ensure that the candidate linearization point is executed at most once along every execution path.
- Finally, at the method's return point(s), it inserts the following check:

  $$\texttt{assert(lres==res} \vee \texttt{(lres==UNDEF} \wedge \texttt{can\_return[res]))}$$

  where `res` is the variable storing the concrete method's return value. This check ensures that either an effectful linearization point has occurred and that the method returned the same result as its specification, or that no effectful linearization point has occurred, but there has been a pure linearization point whose return value matches the concrete return value.

The instrumented operations are validated by calling Verify. Verify takes as arguments the library's instrumented constructor (iop_{init}), its instrumented operations ($iop_1, \ldots, iop_n$), and one command per operation that is to be inserted at each point during the execution of that operation. These are the just the previously computed pure linearization checkers: $check_1, \ldots, check_n$. Note that these checkers have to be passed as arguments to Verify (and cannot simply be instrumented in the source code of the operations), because we want to allow linearization points of pure executions to reside in the code of other threads. To handle this case, Verify also inserts the checkers in its abstractions of the other threads' behaviour. This instrumentation cannot be done statically before calling Verify because these abstractions have not yet been computed.

Verify constructs the most general client of the library and uses an automatic static analysis to prove that the library is memory safe and that the assertions

Algorithm 2. Adaptation of STABILIZE(S, *Rely*) within op_i with checker $check_i$.

1: $S \leftarrow$ SYMB-EXEC($S, \emptyset, check_i$)
2: **repeat**
3: $S_{\text{old}} \leftarrow S$
4: **for all** $(R \mid P \rightsquigarrow Q) \in Rely$ **do**
5: $S \leftarrow S \vee$ ABSTRACT(SYMB-EXEC(MAY-SUBTRACT(S, P, R) $* Q, \emptyset, check_i$))
6: **until** $S = S_{\text{old}}$
7: **return** S

in any `assert` statements in the library are always satisfied. The most general client is a top-level program which models all possible usages of the library. It consists of the constructor followed by an unbounded parallel composition of threads, each of which non-deterministically executes one of public methods of the library in a loop. So, if so assertion violations occur for the most general client of the library, then no library assertion violations will occur for *any* client of the library.

4.4 Soundness

To prove soundness of our algorithm, we first show that the instrumentation described in §4.2 and §4.3 implies linearizability:

Theorem 1 (Instrumentation Correctness). *If a library A is instrumented as described in §4.2 and §4.3 with respect to the specification B, and an execution of a client of the instrumented library did not violate any of assertions, then that execution was linearizable.*

The proof of this theorem is quite technical and can be found in the technical report [21]. Briefly, for each operation, we can pick the instant when `lres` was set as its linearization point, or if `lres` was never set, then the point when `can_return`$[r]$ was first set to true, where r is the eventual return value of the operation.

The soundness of PROVELINEARIZABLE follows directly from Theorem 1 and the specification of VERIFY, which ensures that no library assertions are violated under any execution of any valid client of the library.

Theorem 2 (Soundness). *If calling* PROVELINEARIZABLE *with the arguments* $(init, init\_spec, op_1, spec_1, \ldots, op_n, spec_n)$ *returns 'Success,' then the library consisting of the constructor init and methods* $op_1, \ldots, op_n$ *is linearizable with respect to its specification (init_spec,* $spec_1, \ldots, spec_n$*).*

4.5 Implementation

We have implemented the algorithm for proving linearizability within CAVE, an automatic verification tool for concurrent algorithms based on RGSep. We

take VERIFY to be the RGSep action inference algorithm [20], adapted to execute the corresponding pure checker, $check_i$, symbolically at every step of the 'stabilization' calculations within each instrumented operation, iop_i. Formally, we have changed the implementation of STABILIZE$(S, Rely)$ [20, Alg. 1] to the version shown in Alg. 2. The only changes are the two calls to symbolic execution, which effectively means that VERIFY simulates the pure checker after every atomic command of the current thread accessing the shared state and also after every atomic command of other concurrently executing threads.

Return Set Abstraction. To ensure that VERIFY terminates, abstraction must (under-)approximate the set of values v for which `can_return`$[v]$ is true. While this may seem unnecessary for `tryDequeue` because its pure executions can return only `EMPTY`, it is crucial for specifications, such as `peek` on a stack or a queue, whose pure executions can return an unbounded number of different answers. Our static analyser abstracts over this set by remembering only which program variables and program constants are contained in the set. As there is only a finite set of variables and constants appearing in the input program, the range of this abstraction is finite, and hence the termination of the underlying static analysis is not affected. Formally, this is an instance of 'canonical abstraction' [16] and is analogous to the abstraction performed for pointers.

Implementation Optimisations. Before executing Alg. 1, CAVE first executes VERIFY$(init, op_1, \texttt{skip}, \ldots, op_n, \texttt{skip})$ to check that the uninstrumented library is memory safe: that it does not dereference any invalid pointers and that it does not violate any assertions. The purpose of this initial call is threefold:

1. It aids debugging. If action inference cannot verify that the uninstrumented program is safe (either because the program is erroneous, or because the analysis is imprecise), there is no way that it will succeed in verifying the instrumented programs. Thus, it is better to fail quickly, and give a simpler error message to the user.
2. It can help quickly prune the search space of linearization point assignments. Action inference distinguishes updates to shared memory locations from updates to thread-local data, as only the former have an action associated with them. Thus, we can ignore any candidate linearization point assignments that involve thread-local accesses.
3. The set of RGSep actions inferred by this phase can then used as a starting point for the following VERIFY calls within Alg. 1, thereby making later action inference calls reach their fix-point in a single iteration.

We can further optimise the call to VERIFY in Alg. 1 in two ways. First, it can fail immediately if the correlation between the abstract state and the concrete state is lost. This allows us to fail much more quickly on erroneous linearization point assignments. Second, it first tries to prove linearizability by inlining the instrumented checkers only within the source code of the current thread (i.e., only at the beginning of every stabilization), and if that fails to establish linearizability, then also after every stabilization iteration. This alleviates the cost

Data structure	Lines	Ops	Eff	Pure	LpO	Time(s)
DCAS stack	52/100	2/8	2/4	1/5	0/0	0.1/0.3
Treiber stack [17]	52/100	2/8	2/4	1/5	0/0	0.1/0.3
M&S two-lock queue [14]	54/85	2/4	2/3	1/2	0/0	2.0/16.5
M&S non-block. queue [14]	82/127	2/4	2/3	1/2	0/0	1.7/4.9
DGLM non-block. queue [5]	82/126	2/4	2/3	1/2	0/0	1.8/7.6
Pessimistic set [12]	100	3	2	3	0	247.8
V&Y DCAS-based set [22]	101	3	2	3	0	51.0
ORVYY lazy set [15]	94	3	2	3	1	521.5

Fig. 3. Verification statistics for a collection of stack, queue, and set benchmarks

of inserting the pure checkers within the abstraction of other threads, when this is not needed to prove linearizability.

5 Experimental Evaluation

We have successfully applied Cave to a number of practical concurrent stack, queue, and set algorithms from the literature, which are reported in Fig. 3. For some algorithms, we have considered two versions: one being just core algorithm as normally published, and one being a mostly straightforward extension of the algorithm providing supplementary operations. We present our results for both versions in the same line separating the corresponding numbers with a slash. For each algorithm, we record the number of lines of code excluding comments, blank lines, and the specifications (**Lines**), the number of public methods of the library (**Ops**), the number of effectful methods (**Eff**), the number of methods with pure executions (**Pure**), the number of methods with linearization points in other threads (**LpO**) and the total verification time in seconds (**Time**).

Stack & Queue Benchmarks. The stack algorithms use non-blocking synchronisation, performing a DCAS or a CAS to update the top of the stack. The basic versions of the stack algorithms provide just `push` and `tryPop` operations. The `tryPop` operation has a pure execution in case the stack was empty, in which case it returns a special value (similar to the `tryDequeue` of Fig. 1). The extended implementations also provide `waitPop` (which blocks if the stack is empty), `tryPeek`, `waitPeek`, `waitEmpty`, `isEmpty`, `makeEmpty`.

The queue algorithms support `enqueue` and `tryDequeue` operations with the specifications shown in Fig. 1. The extended versions have two further operations: a blocking dequeue and an emptiness test. The first algorithm is a lock-based design due to Michael and Scott that uses a different lock to protect each end of the queue. The second one is due to the same authors and was presented in Fig. 2. The DGLM queue is a variant of M&S non-blocking queue that was proposed by Doherty et al. [5] and verified in the PVS theorem prover.

Set Benchmarks. These have three operations: adding an element to the set, removing one element from the set, and testing for membership in the set. The

first two operations are effectful, but have pure executions whenever the item to be added (resp. removed) was already in the set (resp. not in the set). In all cases, the set is represented as a sorted singly linked list with two sentinel nodes.

The pessimistic set has a lock per list node, acquired in a hand-over-hand fashion. The V&Y DCAS-set [22] traverses the list optimistically (i.e., with no synchronisation) and then validates that the traversal was correct. The ORVYY lazy set [15] also performs optimistic traversals and uses a bit for marking nodes that are about to be deleted. This allows it to have an efficient wait-free `contains` implementation. The ORVYY lazy set is particularly interesting, because one of the linearization points of `contains` lies within code of a different thread.

We have also run Cave on two further set algorithms: the V&Y CAS-based set [22] and the HHLMSS lazy set [12, §9.7], but it failed to prove linearizability. Verification of the first example failed because one of the calls to Verify timed out, probably due to the current naïve axiomatisation of sorted sequences in the analyser. In the second algorithm, the correct abstraction map lies outside of the abstract domain of our implementation of Verify and, hence, was not be found.[2]

Discussion. From the verification times, one can observe that the stack algorithms are relatively easy to verify. This is because of the rather simple data structure invariants (e.g. the stack is represented by a null-terminated singly-linked list) that Verify has to infer. In contrast, the set algorithms have much more complicated data structure invariants (e.g. the set being represented by a sorted list with special sentinel nodes and there can be multiple arbitrarily long chains of deleted nodes pointing into the sorted list), which take significantly more effort to infer. In all these algorithms, the search space for the effectful candidate linearization point assignments was quite small. For the more complicated examples, searching for incorrect assignments took a small fraction of the whole verification time. The verification time was dominated by the validation of the correct linearization point assignment.

Since our tool relies on abstract interpretation, our verification procedure is incomplete: it is unable to verify many correct programs that lie outside its domain (such as the aforementioned HHLMSS lazy set), and does not provide concrete counterexample traces when the verification fails. Moreover, Cave cannot prove linearizability of effectful executions whose linearization points are inside the code of different threads (such as RDCSS and the elimination-based stack), unless these linearization points are somehow annotated by the programmer. It can, however, prove linearizability of method executions having linearization points within different threads, provided that these executions are logically effect-free, as was the case with the ORVYY lazy set.

[2] The abstraction map for the HHLMSS lazy set is the set of the values of unmarked nodes that are reachable from the head of the list. In contrast, the ORVYY lazy set has a simpler abstraction map: it is the set of the values of all the nodes that are reachable from the head of the list. While it is plausible to extend the analyser to infer such complicated abstraction maps automatically, it is probably better to leave them as input by the programmer.

The main observation of this paper that enabled these verification results was to distinguish executions of the abstract operations (i.e., the specifications) that are pure from those that are effectful. This is related to Elmas et al. [6], who in the context of runtime refinement-violation detection treat operations with a pure specification differently than ordinary operations. Flanagan et al. [8] also had a somewhat related concept of purity, but in their work there is no notion of an abstract operation, and purity is applied only to the implementation. None of the algorithms verified here could have been verified with brute-force search for linearization points.

6 Conclusion

This paper presented a practical technique for automatically proving linearizability. This was implemented in a tool, Cave, which expects a library to be verified together with its atomic functional correctness specification and attempts to prove that the library is linearizable with respect to its specification. We have applied our tool to a number of concurrent stack, queue, and set algorithms, some of which were mechanically verified for the first time.

As this is the first automatic technique for verifying functional correctness of non-trivial concurrent programs, there are several ways in which it can be improved. One such way would be to deal with effectful linearization points in other threads that are 'similar' to a linearization point in the thread being verified (where two program statements are deemed 'similar' if they are abstracted by the same RGSep action). More practically, our prover should be combined with lightweight methods for proving atomicity (e.g., [8]) and with testing techniques for eliminating incorrect linearization point assignments quickly. Further, as such provers become increasingly sophisticated, it will be important to generate proof objects that can be independently checked by a trusted computer program. Last, but not least, there is a never-ending challenge in devising more powerful and more efficient abstract domains for the underlying static analyses used in procedures such as Verify. In particular, improving the support for arrays would enable us to reason about several more concurrent algorithms, such as concurrent hash tables.

Acknowledgements. I would like to thank the anonymous reviewers, Byron Cook, and Peter Sewell for their useful feedback that helped improve the paper. This work was supported by EPSRC grant EP/F036345.

References

1. Abadi, M., Lamport, L.: The existence of refinement mappings. Theoretical Computer Science 82(2), 253–284 (1991)
2. Amit, D., Rinetzky, N., Reps, T., Sagiv, M., Yahav, E.: Comparison under abstraction for verifying linearisability. 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, S.: Thread quantification for concurrent shape analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 399–413. Springer, Heidelberg (2008)
4. Burckhardt, S., Dern, C., Tan, R., Musuvathi, M.: Line-up: a complete and automatic linearizability checker. In: PLDI 2010. ACM, New York (2010)
5. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: de Frutos-Escrig, D., Núñez, M. (eds.) FORTE 2004. LNCS, vol. 3235, pp. 97–114. Springer, Heidelberg (2004)
6. Elmas, T., Tasiran, S., Qadeer, S.: VYRD: verifying concurrent programs by runtime refinement-violation detection. In: PLDI, pp. 27–37. ACM, New York (2005)
7. Filipović, I., O'Hearn, P.W., Rinetzky, N., Yang, H.: Abstraction for concurrent objects. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 252–266. Springer, Heidelberg (2009)
8. Flanagan, C., Freund, S.N., Qadeer, S.: Exploiting purity for atomicity. IEEE Trans. Software Eng. 31(4), 275–291 (2005)
9. Harris, T., Fraser, K., Pratt, I.A.: A practical multi-word compare-and-swap operation. In: Malkhi, D. (ed.) DISC 2002. LNCS, vol. 2508, pp. 265–279. Springer, Heidelberg (2002)
10. Hendler, D., Shavit, N., Yerushalmi, L.: A scalable lock-free stack algorithm. In: SPAA 2004, pp. 206–215. ACM, New York (2004)
11. Herlihy, M.P., Wing, J.M.: Linearizability: a correctness condition for concurrent objects. ACM TOPLAS 12(3), 463–492 (1990)
12. Herlihy, M.P., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann, San Francisco (2008)
13. Liu, Y., Chen, W., Liu, Y.A., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
14. Michael, M., Scott, M.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC 1996. ACM, New York (1996)
15. O'Hearn, P.W., Rinetzky, N., Vechev, M.T., Yahav, E., Yorsh, G.: Verifying linearizability with hindsight. In: PODC 2010. ACM, New York (2010)
16. Sagiv, S., Reps, T.W., Wilhelm, R.: Parametric shape analysis via 3-valued logic. In: POPL 1999, pp. 105–118. ACM, New York (1999)
17. Treiber, R.K.: Systems programming: Coping with parallelism. Tech. report RJ5118, IBM Almaden Res. Ctr. (1986)
18. Vafeiadis, V.: Modular fine-grained concurrency verification. PhD thesis. Tech. report UCAML-CL-TR-726, Univ. of Cambridge Computer Laboratory (2007)
19. Vafeiadis, V.: Shape-value abstraction for verifying linearizability. In: Jones, N.D., Müller-Olm, M. (eds.) VMCAI 2009. LNCS, vol. 5403, pp. 335–348. Springer, Heidelberg (2009)
20. Vafeiadis, V.: RGSep action inference. In: Barthe, G., Hermenegildo, M. (eds.) VMCAI 2010. LNCS, vol. 5944, pp. 345–361. Springer, Heidelberg (2010)
21. Vafeiadis, V.: Automatically proving linearizability (long version). Tech. report UCAML-CL-TR-778, Univ. of Cambridge Computer Laboratory (2010)
22. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: Gupta, R., Amarasinghe, S.P. (eds.) PLDI, pp. 125–135. ACM, New York (2008)

Model Checking of Linearizability of Concurrent List Implementations*

Pavol Černý[1], Arjun Radhakrishna[1], Damien Zufferey[1],
Swarat Chaudhuri[2], and Rajeev Alur[3]

[1] IST Austria
[2] Pennsylvania State University
[3] University of Pennsylvania

Abstract. Concurrent data structures with fine-grained synchronization are notoriously difficult to implement correctly. The difficulty of reasoning about these implementations does not stem from the number of variables or the program size, but rather from the large number of possible interleavings. These implementations are therefore prime candidates for model checking. We introduce an algorithm for verifying linearizability of singly-linked heap-based concurrent data structures. We consider a model consisting of an unbounded heap where each vertex stores an element from an unbounded data domain, with a restricted set of operations for testing and updating pointers and data elements. Our main result is that linearizability is decidable for programs that invoke a fixed number of methods, possibly in parallel. This decidable fragment covers many of the common implementation techniques — fine-grained locking, lazy synchronization, and lock-free synchronization. We also show how the technique can be used to verify optimistic implementations with the help of programmer annotations. We developed a verification tool CoLT and evaluated it on a representative sample of Java implementations of the concurrent set data structure. The tool verified linearizability of a number of implementations, found a known error in a lock-free implementation and proved that the corrected version is linearizable.

1 Introduction

Concurrency libraries such as the java.util.concurrent package JSR-166 [13] or the Intel Threading Building Blocks support the development of efficient multithreaded programs by providing *concurrent data structures*, that is, concurrent implementations of familiar data abstractions such as queues, sets, and stacks. Many sophisticated algorithms that use lock-free synchronization have been proposed for this purpose (see [10] for an introduction). Such implementations are not race-free in the classic sense because they allow concurrent access to shared memory locations without using locks for mutual exclusion. This also makes them notoriously hard to implement correctly, as witnessed by several bugs found in

* This research was partially supported by NSF grants CCF 0905464 and CAREER Award 0953507 and by the Gigascale Systems Research Center.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 465–479, 2010.

published algorithms [5,16]. The complexity of such algorithms is not due to the number of lines of code, but due to the multitude of interleavings that must be examined. This suggests that such applications are prime candidates for formal verification, and in particular, that *model checking* can be a potentially effective technique for analysis.

A typical implementation of data structures such as queues and sets consists of a linked list of vertices, with each vertex containing a data value and a next pointer. Such a structure has two distinct sources of infinity: the data values in individual vertices range over an unbounded domain, and the number of vertices is unbounded. A key observation is that methods manipulating data structures typically access data values in a restricted form using only the operations of equality and order. This suggests that the contents of a list can be modeled as a *data word*: given an unbounded domain D with equality and ordering, and a finite enumerated set Σ of symbols, a data word is a finite sequence over $D \times \Sigma$. In our context, the set D can model keys used to search through a list, the ordering can be used to keep the list sorted, and Σ can be used to capture features such as marking bits or vertex-local locks used by many algorithms. However, when concurrent methods are operating on a list without acquiring global locks, vertices may become inaccessible from the head of the list. Indeed, many bugs in concurrent implementations are due to the fact that "being a list" is not an invariant, and thus, we need to explicitly model the next pointers and the shapes they induce (see Figure 1). In this paper, we propose a formal model for a class of such algorithms, identify restrictions needed for decidability of linearizability, and show that many published algorithms do satisfy these restrictions.

We introduce the model of *singly-linked data heaps* for representing singly-linked concurrent data structures. A singly-linked data heap consists of a set of vertices, along with a designated start vertex, where each vertex stores an element of $D \times \Sigma$ and a next field that is either null or a pointer to another vertex. Methods operating on such structures are modeled by *method automata*. A method automaton has a finite internal state and a finite number of pointer variables ranging over vertices in the heap. The automaton can test equality of pointers and equality as well as ordering of data values stored in vertices referenced by its pointer variables. It can update fields of such vertices, and update its pointer variables, for instance, by following the next fields. The model restricts the updates to pointers to ensure that the list is traversed in a monotonic manner from left to right. We show that this model is adequate to capture operations such as search, insert, and delete, implemented using a variety of synchronization mechanisms, such as fine grained vertex-local locking, lazy synchronization, and primitives such as compare-and-set.

Our main result is the decidability of linearizability of method expressions. A method expression allows to combine a fixed number of method automata using sequential and parallel composition. Linearizability [11] is a central correctness requirement for concurrent data structure implementations. Our algorithm takes as input a precondition I in addition to a method expression E and checks that all executions of E starting from a heap that satisfies I are linearizable. For

example, given two methods to insert and delete elements of a list, our decision procedure can check whether every execution of the parallel composition of the two methods that starts from a sorted list is linearizable. Our decidability proof is developed in two steps.

First, we show how linearizability of a method expression E can be reduced to a reachability condition on a method automaton A. The automaton A simulates E and all of its possible linearizations. For instance, if E is the $A_1 \parallel A_2$ then the possible linearizations are $A_1; A_2$ and $A_2; A_1$. The principal insight in the construction of A is that the automata in E can proceed through the list almost in a lock-step manner. This result assumes that the methods we analyze are deterministic when run sequentially. Note that the assumption is satisfied by all the implementations we analyzed.

Second, we show that reachability for a single method automaton is decidable: given a method automaton, we want to check if there is a way to invoke the automaton so that it can reach a specified state. We show that the problem can be reduced to finite state reachability problem. The main idea is that one need not to remember values in D, but only the equality and order information on such values.

We implemented a tool CoLT (short for Concurrency using Lockstep Tool) based on the decidability results. The tool implements only the case of the parallel composition of two method automata. We evaluated the tool on a number of implementations, including one that uses hand-over-hand vertex local locking, one that uses an optimistic approach called lazy synchronization, and one that uses lock-free synchronization via compare-and-set. All of these algorithms are described in [10] and the Java source code was taken from the book's website. The tool verified that the fine-grained and lazy algorithms are linearizable, and found a known bug in the remove method of the lock-free algorithm. The tool allows the user to provide linearization points, which reduces search space significantly. The experiments show that our techniques scale to real implementations of concurrent sets. The running times were under a minute for all cases of fine-grained and lazy methods (even without linearization points), and around ten minutes for lock-free methods (when the programmer specified linearization points).

Related Work. Verifying correctness of concurrent data structures has received a lot of attention recently. A number of machine-checked manual proofs of correctness exists in the literature [7,19]. We know of two model checking approaches ([14,20]). Both these works consider only a bounded heap, whereas our approach does not impose any bound on the size of the heap. Static analysis methods based on shape analysis [3,18] require user-specified linearization points. In contrast, our approach does not need linearization points. The user has an option to provide linearization points to improve performance. The experiments show that this is not necessary at all for e.g. the fine-grained locking, and lazy list implementations. Furthermore, shape analysis approaches are sound, but not complete techniques, whereas our algorithm is both sound and complete for a bounded number of threads. As for the model of the heap, closest to ours is the model

of [1], but the work in [1] is on abstraction of sequential heap accessing programs. There is an emerging literature on automata and logics over data words [17,4] and algorithmic analysis of programs accessing data words [2]. While existing literature studies acceptors and languages of data words, we want to handle destructive methods that insert and delete elements.

2 Singly-Linked Data Heaps and Method Automata

Singly-Linked Data Heaps. Let D be an unbounded set of data values equipped with equality and linear order $(D, =, <)$ and let Σ be a finite set of symbols. A *singly-linked data heap* is a tuple $(V, next, flag, data, h)$, where V is a finite set of vertices, *next* is a partial function from V to V, *flag* is a function from V to Σ, *data* is a function from V to D, and $h \in V$ denotes the initial vertex.

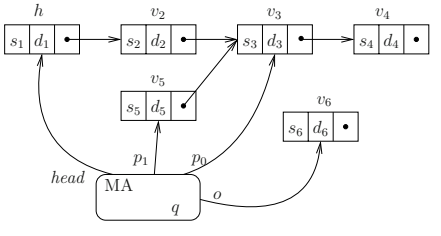


Fig. 1. Singly-linked data heap and a method automaton

The structure L can be naturally viewed as a labeled graph with edge relation *next*. L is *well-formed* if this graph has no cycles reachable from h. For each well-formed heap L as above, we define a finite data word (over $\Sigma \times D$) represented by the list starting at h. Figure 1 shows a singly-linked data heap with six vertices that contain values from Σ and D which define the data word $(s_1, d_1)(s_2, d_2)(s_3, d_3)(s_4, d_4)$.

Method automata: Syntax. A *method automaton* is a tuple $(Q, P, DV, T, q_0, F, head, O)$, where Q is a finite set of states, P is a finite partially-ordered set of pointer variables, DV is a finite set of data variables, T is a set of transitions (as explained below), $q_0 \in Q$ is the initial state, $F \subseteq Q$ is a set of final states, *head* is a pointer constant, and O is a set of pointer constants.

A method automaton operates on a singly-linked data heap $L = (V, next, flag, data, h)$. The pointer variables range over $V \cup \{nil\}$, where *nil* is a special value, and are denoted by e.g. p, p_0, p_1. Let $\leq_P$ be the partial order on P. The partial order is required to have a minimum element, denoted by p_0. The variable p_0 is called the *current pointer*, and the other variables in P are called *lagging pointers*. The constant *head* points to the vertex h and is shared across method automata. The pointer constants in the set O (denoted by e.g.o, o_0, o_1) give method automata input/output capabilities and are referred to as *IO pointers*. The set R of pointers (i.e. pointer variables and pointer constants) of a method automaton is defined by $R = P \cup \{head\} \cup O$. The data variables in DV range over the unbounded domain D.

The set of transitions T is a set of tuples of the form (q, G, A, q'), where $q, q' \in Q$ are states, G is a guard, and A is an action. There are no outgoing transitions from the final states.

Let $succ_P$ be the successor relation defined by the partial order $\leq_P$. The syntax of *guards* G and *actions* A are now defined as:

$$
\begin{array}{rl}
DE & ::= v \mid data(p) \\
G & ::= flag(p) = s \quad (\text{where } s \in \Sigma) \mid DE = DE \mid DE < DE \mid p = p' \\
& \quad \mid p = nil \mid p = next(p') \mid next(p) = nil \mid G \text{ and } G \mid \neg G \mid true \\
Act & ::= flag(p) := s \quad (\text{where } s \in \Sigma) \mid data(p) := DE \\
& \quad \mid next(p) := nil \mid next(p) := p' \quad (\text{where } succ_P(p', p)) \\
& \quad \mid values(p) := (s, DE, p') \quad (\text{where } succ_P(p', p)) \\
& \quad \mid v := DE \mid p := p' \quad (\text{where } succ_P(p', p)) \\
& \quad \mid p := nil \mid p_0 := next(p_0).
\end{array}
$$

where p, p' are pointer variables, p_0 is the current pointer (minimum pointer variable), and v is a data variable.

The guards include symbol, data and pointer comparison and their boolean combinations. The restriction $succ_P(p', p)$ placed on some actions enforce that the heap is traversed in a monotonic manner. This necessitates that pointer variables are statically ordered, and the furthest pointer can be assigned to the next of its vertex, but lagging pointers can be assigned only to a pointer further up in this ordering. Fields of vertices, including the next field, corresponding to lagging pointers can be updated. Also, the three fields of vertices (Σ value, data value, and the next pointer) can be updated together atomically (this is needed for encoding some of the Java concurrency primitives).

We require the actions of a method automaton to satisfy a restriction *OW*, abbreviation for "*One write before move.*" This restriction states that there is at most one action modifying $flag(p)$, at most one action modifying $data(p)$, and at most one action modifying $next(p)$ performed between two successive changes of the value of the pointer variable p. The restriction can be enforced syntactically — we omit the details. We note that the restriction OW holds for every implementation we have encountered and that we show that without this restriction, the linearizability problem becomes undecidable.

A method automaton is *deterministic* iff given a state and a valuation of variables, at most one guarded action is enabled.

Figure 1 shows a method automaton in state q. Its *head* pointer points to the vertex h of the heap. A client of the automaton can store values in the vertex v_6 pointed to by the IO pointer o. The variables p_0 and p_1 are pointer variables of the method automaton.

Examples. We illustrate the model by showing how the model captures synchronization primitives and other core features of concurrent data structure algorithms.

- *Traversing a list.* Let us suppose we want the current pointer p_0 to traverse a list (assumed to be sorted) until it finds a data value equal or larger to the one stored at a vertex pointed to by an IO pointer o. A method automaton can achieve this by having a transition such as: $(q, data(p_0) < data(o), p_0 := next(p_0), q)$.

- *Inserting a vertex.* Assume that the position to insert the vertex has been found - the new vertex o is to be inserted between p_1 and p_0. The transition relation can then include $(q, \mathit{true}, \mathit{next}(o) := p_0, q_1)$ and $(q_1, \mathit{true}, \mathit{next}(p_1) := o, q_2)$.
- *Locking individual vertices.* We can model locking of vertices using the Σ value. Let us suppose that $\Sigma = \{u, l_1, l_2, \ldots\}$, for unlocked, locked by thread 1, locked by thread 2, etc. Locking is captured by the transition: $(q_0, \mathit{flag}(p) = u, \mathit{flag}(p) := l_1, q_1)$ for thread number 1. Unlocking can be modeled as follows: $(q_1, \mathit{flag}(p) = l_1, \mathit{flag}(p) := u, q_2)$.
- *Modeling compare-and-set.* The synchronization operation compare-and-set is supported by several contemporary architectures as well as Java Concurrency library. The operation takes two arguments, an expected value (ev) and an update value (uv). If the current value of the register (for hardware) or a reference (in Java) is equal to the expected value, then it is replaced by the update value. The operation returns a Boolean indicating whether the value changed. The operation is modeled by the following transition tuples: $(q, \mathit{data}(p) = ev, \mathit{data}(p) := uv, q')$ and $(q, \mathit{data}(p) \neq ev, -, q'')$.

Semantics. An *automaton invocation* $A(L, io)$ consists of a method automaton $A = (Q, P, DV, T, q_0, F, \mathit{head}, O)$, a singly-linked data heap $L = (V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$, and a function $io : O \to V$. The pair (L, io) is the *method input*. A method input is *well-formed* if L is well-formed, and for all variables $o \in O$, we have that the vertex $io(o)$ is unreachable from h and $\mathit{next}(io(o))$ is undefined. A method automaton is initialized by having its *head* pointer pointing to h and its input variables in O initialized by the function io. The output of a method is also realized via the variables O, which are shared with the client.

The semantics is given by the transition system denoted by $[\![A(L, io)]\!]$ for a well-formed input (L, io). The definition formalizes the following intuition: a transition of the method automaton is chosen nondeterministically and executed atomically. Let us use a special value *nil* to model the null pointer, and let $q_{err} \notin Q$ be a special state reached on null-pointer dereference. Let $L = (V, \mathit{next}, \mathit{flag}, \mathit{data}, h)$ and $A = (Q, P, DV, T, q_0, F, \mathit{head}, O)$. A configuration $s = (L, q, U, dv)$ of $[\![A(L, io)]\!]$ has four components: a heap L, a state q in $Q_{err} = Q \cup \{q_{err}\}$, a valuation of pointers $U : R \to V \cup \{nil\}$ and a valuation of data variables $dv : DV \to D$. A configuration is *initial* if it is of the form (L, q_0, U, dv), where U sets all pointer variables to h and dv sets all the data variables to the value $\mathit{data}(h)$. Note that there is a unique initial configuration in $[\![A(L, io)]\!]$.

The transition relation of $[\![A(L, io)]\!]$ is defined as expected. For example, if $(q, \mathit{true}, p := \mathit{next}(p), q')$ is a transition of the method automaton A, then there is a transition from a configuration (L, q, U, dv) to (L, q', U', dv), where $U'(p') = U(p')$ for all $p' \in R$ such that $p' \neq p$ and $U'(p) = U(\mathit{next}(p))$. The relation $(L, io) \xrightarrow{A} (L', io')$ is defined to hold if there exists a path from the

initial configuration of $[\![A(L, io)]\!]$ to a configuration (L', q, U, dv), where q is a final state and io' is a restriction of U to IO pointers.

Composition of method automata. Consider two method automata A_1 and A_2. We define the *parallel composition* $A_1 \parallel A_2$ informally by describing the semantics. The state space of the parallel composition of A_1 and A_2 with IO pointers $io_1 \cup io_2$ is the product of the state space of $[\![A_1(L, io_1)]\!]$ and $[\![A_2(L, io_2)]\!]$, with the singly-linked data heap L being shared between the two automata. The transition function defines interleaving semantics. We omit further details in the interest of space. We analogously define *sequential composition* $A_1 ; A_2$. *Method expressions* compose a finite set of method automata sequentially and in parallel, they are defined by the following grammar rules: $E ::= E_S \mid (E_S \parallel E)$ and $E_S ::= A \mid (A ; E_S)$, where A is a method automaton. The semantics is given by the transition system $T(E, L, io)$ and the relation $(L, io) \xrightarrow{E} (L', io')$ is defined as in the case of single automata. Given a method expression E let $Aut(E)$ be the set of method automata that are components of E.

3 Verifying Linearizability

Linearizability [11] is the standard correctness condition for concurrent data structure implementations. In this section, we study the linearizability problem for method expressions. The proofs omitted here are available in [6].

A *history* is a sequence of method invocations and method returns (a pair of method invocation and corresponding return is called a *method call*). We say that a history h is a history of a method expression E, if h corresponds to an execution of E. A *sequential history* is such that a method invocation is immediately followed by the corresponding method return. A history is *complete* if each method invocation is followed (not necessarily immediately) by a method return. Intuitively, a sequential history h_s is a *linearization* of a complete history h, if for all threads, the projection of h to a thread is the same as the projection h_s to the same thread, and the following condition holds: if a method call m_0 precedes method call m_1 in h, then the same is true in h_s. We omit further details for lack of space, and we refer the reader to [11,10] for a formal definition, as well as for a definition of linearizations of histories that are not complete.

A method expression is *sequential*, if it does not contain any parallel composition. Note that given a sequential method expression E_s, there is a unique complete history of E_s, denoted by $hist(E_s)$, which calls all the automata in $Aut(E_S)$. Given a method expression E and a history h of E, let $Seq(E, h)$ be the set of sequential method expressions E_s such that $hist(E_s)$ is a linearization of h. For example consider the method expression $E = E_1 \parallel E_2 \parallel E_3$ and an execution h of E where E_3 starts only after E_2 has finished and the execution of E_1 overlaps with both E_2 and E_3. The set $Seq(E, h)$ is $\{(E_1 ; E_2 ; E_3), (E_2 ; E_1 ; E_3), (E_2 ; E_3 ; E_1)\}$. Let $Seq(E)$ denote the set of sequential method expressions E_s such that $Aut(E) = Aut(E_S)$. Note that we always have $Seq(E, h) \subseteq Seq(E)$.

For a method expression E, a history h of E, a well-formed input (L, io), a heap L' and a function io', we write $(L, io) \xrightarrow{E,h} (L', io')$ if a node corresponding to (L', io') is reached in $T(E, L, io)$ using an execution whose history is h.

We have now defined the notions we need to state the definition of linearizability. However, it is often useful to specify a condition under which we are interested in checking linearizability. Such preconditions can be defined using acceptors — method automata that do not modify the heap. An example precondition is that the data values in the list starting in the initial node are sorted.

A method automaton I is called an *acceptor* if it does not use the commands that modify the heap (the first five actions defined by the grammar in Section 2): Given an acceptor I, and a well-formed input (L, io), I *accepts* (L, io) (denoted by $I \models (L, io)$) if there exists (L', io') such that $(L, io) \xrightarrow{I} (L', io')$.

We now define an equivalence relation on singly-linked data heaps. Two singly-linked data heaps are equivalent when they represent the same value of an abstract data type. As an example, we consider sets of elements of the data domain D as the abstract data type. A list can represent a set containing data values from unmarked nodes (marking is represented by Σ-values). Two heaps are then equivalent if the unmarked elements they contain are the same.

A method automaton is an *adt-checker* if it is a deterministic method automaton with no IO pointers. Given an *adt*-checker C, two heaps L_1 and L_2 are equivalent ($L_1 \equiv_C L_2$), if there exists a heap L' such that $L_1 \xrightarrow{C} L'$ and $L_2 \xrightarrow{C} L'$. The relation $\equiv_C$ is extended to pairs (L, io) as follows: $(L_1, io_1) \equiv_{C,b} (L_2, io_2)$ iff $L_1 \equiv_C L_2$, b is a bijection between the domains of io_1 and io_2 and we have $io_1(o) = io_2(b(o))$. We omit b if it is clear from the context, for instance when comparing different compositions of the same automata.

We are now ready to state the central definition of this paper:

> Given an acceptor I and an *adt*-checker C, a method expression E is (I, C)-*linearizable* if and only if the following condition holds: for all L, io, L_P, io_P, h such that (L, io) is a well-formed input, $I \models (L, io)$, we have that if $(L, io) \xrightarrow{E,h} (L_P, io_P)$, then there exists a sequential method expression E_s in $Seq(E, h)$ and L_S, io_S such that $(L, io) \xrightarrow{E_s} (L_S, io_S)$ and $(L_P, io_P) \equiv_C (L_S, io_S)$.

The definition of method expression linearizability captures the standard definition of linearizability [11] for the case of composition of a bounded number of methods. In the standard definition, we have the requirement that all histories (possibly of unbounded length) are linearizable. Method expression linearizability not only checks that all bounded histories of E are linearizable, it also checks that starting on the same list, every interleaved execution should finish with the same list as at least one sequential execution whose history is a linearization of the history of the interleaved execution. Put yet another way, method expression linearizability checks not only that all histories of E are linearizable, but also checks that all histories of P_1 ; E ; P_2 are linearizable, for all sequential programs P_1 and P_2. As an example, consider a set data structure with methods

`insert` and `contains`. With these two methods, the requirement is captured by the history that (starting with the empty list) calls `insert` at the beginning and `contains` at the end of the execution. A formal comparison of the definitions is deferred to the full version.

Decision problem. We now formulate the decision problem considered in this paper:

> Given a method expression E, an acceptor I and an *adt*-checker C the *method expression linearizability problem* is to decide whether E is (I, C)-linearizable.

In the remainder of this paper, we assume that the method expressions E are composed of deterministic method automata. This assumption means that given an expression E, all sequential method expressions in $Seq(E)$ are deterministic.

3.1 Reachability

In order to show that method expression linearizability is decidable, we will need the following results. First, we show that the effect of the method expression can be captured by a single method automaton, which is built using a lockstep construction. Second, we show that reachability is decidable for method automata.

Theorem 1. *Given a method expression E, there exists a method automaton $LS(E)$ such that for all L_P, L'_P, io_P, io'_P such that (L_P, io_P) is a well-formed method input, we have $(L, io) \xrightarrow{E} (L', io')$ iff $(L, io) \xrightarrow{LS(E)} (L', io')$.*

Proof. The idea behind constructing method automaton $LS(E)$ is to update the current pointers of all the method automata in $Aut(E)$ in lockstep manner — i.e. that the current pointers of the automata traverse the list at most one step apart. For example, if the current pointer of A_1 is one step ahead of the current pointers of the other automata, then transitions of the other automata are scheduled until the current pointers point to the same position. At that point, a transition of any automaton can be chosen. The lockstep construction is a partial-order reduction. The construction is complicated by the presence of lagging pointers. The solution consists of nondeterministically guessing the interaction of the automata via lagging pointers. In this step the restriction OW is needed. □

Let $A = (Q, P, DV, T, q_0, F, head, O)$ be a method automaton and $q \in Q$. The *method automaton reachability problem* is to decide whether there exist a well-formed method input (L, io), a heap L', a valuation of pointer variables U, and a valuation of data variables dv such that in the transition system $[\![A(L, io)]\!]$, the configuration (L', q, U, dv) is reachable from the initial configuration.

Theorem 2. *The method automaton reachability problem is decidable. The complexity is polynomial in the number of states of the automaton, and exponential in the number of its pointer and data variables.*

The main insight in the construction is that, as the automaton traverses the heap monotonically from left to right, the information stored in vertices pointed to by lagging pointers that is needed for evaluating guards of the transitions can be encoded in a finite manner. More concretely, one need not to remember values in D, but only the equality and order information on these values.

3.2 Deciding Linearizability

The following theorem is the main result of the paper.

Theorem 3. *The method expression linearizability problem is decidable.*

Proof. The proof is by reduction to reachability in method automata, which in turn reduces (by Theorem 2) to reachability in finite state systems. We show how method expression linearizability can be reduced to reachability in a method automaton. Given an acceptor I, a method expression E and an *adt*-checker C, the method automaton $\mathit{LinCheck}(I, E, C)$ simulates I followed by E followed by C, and compares the results to simulation of I followed by E_s followed by C for all $E_s \in \mathit{Seq}(E)$. $\mathit{LinCheck}(I, E, C)$ reaches an error state if there is an unlinearizable execution of E starting from a heap accepted by I. Given a method expression E, the number of automata $\mathit{LinCheck}(I, E, C)$ simulates grows exponentially with the number of methods in E.

First, we use Theorem 1 to show that instead of simulating $(I\ ;\ E\ ;\ C)$ (resp. $(I\ ;\ E_s\ ;\ C)$), one can simulate the method automaton $\mathit{LS}(I\ ;\ E\ ;\ C)$ (resp. $\mathit{LS}(I\ ;\ E_s\ ;\ C)$). Second, we show how $\mathit{LinCheck}(I, E, C)$ can simulate the automaton $\mathit{LS}(I\ ;\ E\ ;\ C)$ and all the automata $\mathit{LS}(I\ ;\ E_s\ ;\ C)$ for $E_S \in \mathit{Seq}(E)$. on the same input heap and reach an error state if there is an unlinearizable execution of $\mathit{LS}(E)$. The key idea is once again that the current pointers of all the automata can advance in a lockstep manner. The reason is much simpler in this setting than in the proof of Theorem 1 — here the automata do not communicate at all (the only reason we are simulating the the automata together is that they run on the same input heap). $\mathit{LinCheck}(I, E, C)$ reaches an error state if none of the expressions in $\mathit{Seq}(E)$ can simulate $\mathit{LS}(E)$. This is the case when for example a particular position in the output list for $\mathit{LS}(I\ ;\ E\ ;\ C)$ is different from that position in output lists of $\mathit{LS}(I\ ;\ E_S\ ;\ C)$ for all $E_S \in \mathit{Seq}(E)$. Such condition is checkable by a method automaton, so $\mathit{LinCheck}(I, E, C)$ can take a transition to a particular state u if it occurs. The state u is then reachable iff E is not (I, C)-linearizable. We have reduced linearizability to reachability in method automata. We can thus conclude by using Theorem 2. □

Undecidable extensions. The following theorem shows that the restriction OW is necessary for decidability.

Theorem 4. *For method automata without the OW restriction, the method expression linearizability problem is undecidable.*

The proof of this theorem also implies that if we lift the restrictions on how the pointer variables are updated, the method expression linearizability problem becomes undecidable as well.

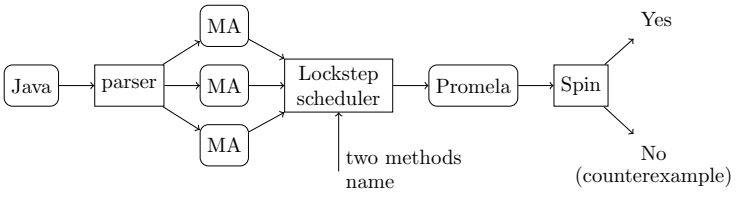


Fig. 2. CoLT Toolchain

4 Experimental Evaluation

4.1 Examples

This section presents a range of concurrent set algorithms where the set is implemented as a linked list whose vertices are sorted by their keys. Each key occurs at most once in the set. The concurrent set provides an interface consisting of three methods: `contains`, `add`, and `remove`. The main difference in the algorithms comes from the synchronization style they use. The synchronization techniques we consider in our experiments are *fine-grained locking*, *optimistic synchronization*, *lazy synchronization*, and *lock-free synchronization*:

- In the *fine-grained locking* approach each vertex is locked separately. During the traversal we use "hand-over-hand" locking, where a vertex is unlocked only after its successor is locked. When an insertion or a deletion is performed, two successive vertices are kept locked.
- A problem with fine-grained locking is that modifications in disjoint parts of the list can still block each other. In *Optimistic* synchronization [10] a thread does not acquire locks as it traverses the list, but only when it finds the part it is interested in. Then the thread needs to re-traverse the list to make sure the locked vertices are still reachable (*validation* phase).
- The *lazy* synchronization algorithm [9] improves the optimistic one in two main aspects. First, the methods do not need re-traversal. Second, `contains` (commonly thought to be the most used method), do not use locks anymore. The most significant change is that the deleted vertices are marked.
- A method is called *lock-free* if delay in one thread executing the method cannot delay other threads executing the method. The lock-free algorithms [8,15] we analyze use the Java `compareAndSet` operation to overwrite values.

4.2 Implementation

The CoLT tool chain can be seen in Figure 2. The input to the tool is a Java file and two method names. The Java methods are parsed into method automata. Then the lockstep scheduler selects the method automata corresponding to the given method names, and produces a finite-state model using the (simplified) construction from the proof of Theorem 3. The finite state model is then checked by the SPIN [12] model checker. If SPIN cannot validate the model, it returns

a counterexample trace that describes an unlinearizable execution. CoLT then gives the programmer a visual representation of the trace.

In the rest of this subsection, we summarize the main issues in translating the Java implementations of concurrent data set algorithms to method automata. We refer the reader to [6] for further details on implementation.

Acceptors and *adt*-checkers. We use an acceptor to assert that the input list is sorted. In the case of the optimistic algorithm we also need an *adt*-checker to handle the marked vertices, i.e. vertices removed logically but not physically. For the other algorithms the *adt*-checker is the identity function.

Phases approach. We implemented a simplified version of the construction from the proof of Theorem 3. It relies on the fact that all the examples we considered work in two phases: in the first phase, a list is traversed without modification (or with limited modification in the case of the lock-free algorithm) and in the second phase, the list is modified "arbitrarily". This simplifies the implementation by reducing the amount of nondeterministic guessing that is necessary, but relies on annotations to identify the phases.

Validate. The optimistic algorithm violates the monotonic traversal restriction as it traverses the list twice, once to find the required vertex and lock it and again to validate that the locked vertex is still accessible from the head of the list. We implemented a heuristic to extend the scope of our tool to cover the optimistic algorithm. For this heuristic, we require annotations in the code that mark the first and the second traversal. Given these annotations, the tool can decompose each method into two method automata, one that finds and locks the vertex and one for validation. A construction similar to sequential composition of these two automata is then used to model an optimistic method.

Retry. The core traversal of fine-grained, lazy and lock-free algorithms is monotonic. The only caveat is that when an operation such as insertion or deletion fails the method might abort and "retry" by setting all pointers to the head, which our restrictions disallows. We emphasize that retry behavior is very different from the validate behavior of the optimistic algorithm. The aborted executions in the fine-grained, optimistic, and lazy methods have no effect on the heap. In the lock-free method, the effect is limited and simply defined. We implemented a simple heuristic to deal with retry behavior. The heuristic produces a method automaton that stops simulating an execution if a retry occurs. One can easily prove for all algorithms we have considered that if the parallel composition of method automata constructed in this way is linearizable iff the original parallel composition is linearizable.

Linearization points. Our tool enables programmers to specify linearization points. Specifying them is not needed, but leads to reduction of the search space, and thus to improving memory consumption and running time of experiments.

Table 1. Experimental results

Algorithm	Methods	M_1 loc/pts	M_2 loc/pts	Lin. points	Depth	Mem (MB)	Time (s)	Res
Fine-grained	remove ‖ contains	29/2	23/2	No	157	10.2	0.85	Yes
Fine-grained	remove ‖ remove	29/2	29/2	No	141	8.3	0.46	Yes
Fine-grained	remove ‖ add	29/2	26/2	No	303	18.1	2.4	Yes
Optimistic	add ‖ remove	40/3	38/3	No	110	37.6	5.86	Yes
Optimistic	contains ‖ contains	30/3	30/3	No	150	37.6	6.9	Yes
Optimistic	remove ‖ remove	38/3	38/3	No	130	36.2	6.35	Yes
Lazy	remove ‖ remove	36/3	36/3	No	164	20.1	2.68	Yes
Lazy	remove ‖ add	36/3	34/3	No	164	26.3	3.51	Yes
Lazy	contains ‖ remove	36/3	6/1	No	136	13.2	1.28	Yes
Lazy	remove_1 ‖ add_1	36/3	34/3	No	137	24.2	3.17	No
Lazy	remove_2 ‖ remove_2	34/3	34/3	No	143	17.9	2.18	No
Lock-free	contains ‖ contains	9/2	9/2	No	98	6.4	0.25	Yes
Lock-free	remove ‖ remove	34/3	34/3	Yes	95	77.6	8.08	No
Lock-free	remCorr ‖ remCorr	34/3	34/3	Yes	268	1908.3	605	Yes
Lock-free	add ‖ remCorr	35/3	34/3	No	?	out	?	?
Lock-free	add ‖ remCorr	35/3	34/3	Yes	267	1550.3	577	Yes
Lock-free	add ‖ contains	35/3	9/2	No	400	18984.1	5700	Yes

4.3 Experiments

We evaluated the tool on the fine-grained, optimistic, lazy, and lock-free implementations of the concurrent set data structure. The Java source code was taken from the companion website to [10]. All the experiments were performed on a server with an 1.86GHz Intel Xeon processor and 32GB of RAM.

The results of the experiments are presented in Table 1. the third (fourth) column contains the number of lines of code and the number of pointer variables of the first (second) method. The fifth column indicates whether linearization points were used. The sixth column lists the maximum depth reached in the exploration of the finite state graph. The last column indicates whether the method expression was linearizable.

First, to evaluate our analysis on implementations of fine-grained locking algorithms, we ran the remove method in parallel with itself, the contains method, and the add method. The memory consumption was under 20MB and the running time under 3s in all cases.

Second, we analyzed the optimistic implementations. The Java file was annotated to use the heuristic described in the previous subsection. CoLT validates the optimistic implementations in under 40MB of memory for every case. The heuristic influence heavily some of the tool's components; hence, the resources consumption of these results are not directly comparable with the others.

Third, we analyzed lazy-synchronization implementations. The tool CoLT verified linearizability in the same cases as the fine-grained locking algorithm. We used the tool to analyze modifications of the add and remove methods suggested as exercises in [10]. One exercise suggests simplification of the validation

check (methods remove_1 and add_1), the other asks using only one lock (method remove_2). We used the tool on $\text{remove}_1 \parallel \text{add}_1$, and on $\text{remove}_2 \parallel \text{remove}_2$. In both cases, CoLT reported these compositions not to be linearizable.

Fourth, we considered lock-free implementations. CoLT found that the parallel composition of remove with itself is not linearizable. This is a known bug, reflected in the online errata for [10]. When we corrected the bug according to the errata (method removeCorr —short for removeCorrected), the tool showed that the parallel composition of remove with itself is linearizable. We observe that the memory usage is larger, for example for the parallel composition of the corrected remove method with the add method, even when the linearization are provided. The tool runs out of memory without the linearization points. The reason is that, compared to the other algorithms, the input list can contain vertices marked for deletion, thus increasing the number of inputs to consider.

5 Conclusion

Summarizing, the main contributions of the paper are two-fold: first, we prove that linearizability is decidable for a model that captures many published concurrent list implementations, and second, we showed that the approach is practical by applying the tool to a representative sample of Java methods implementing concurrent data sets.

References

1. Abdulla, P., Atto, M., Cederberg, J., Ji, R.: Automated analysis of data-dependent programs with dynamic memory. In: Liu, Z., Ravn, A.P. (eds.) ATVA 2009. LNCS, vol. 5799, pp. 197–212. Springer, Heidelberg (2009)
2. Alur, R., Černý, P., Weinstein, S.: Algorithmic analysis of array-accessing programs. In: Grädel, E., Kahle, R. (eds.) CSL 2009. LNCS, vol. 5771, pp. 86–101. Springer, Heidelberg (2009)
3. 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)
4. Bojańczyk, M., Muscholl, A., Schwentick, T., Segoufin, L., David, C.: Two-variable logic on words with data. In: LICS, pp. 7–16 (2006)
5. Burckhardt, S., Alur, R., Martin, M.: Checkfence: checking consistency of concurrent data types on relaxed memory models. In: PLDI, pp. 12–21 (2007)
6. Černý, P., Radhakrishna, A., Zufferey, D., Chaudhuri, S., Alur, R.: Model checking of linearizability of concurrent data structure implementations. Technical Report IST-2010-0001, IST Austria (April 2010)
7. Colvin, R., Groves, L., Luchangco, V., Moir, M.: Formal verification of a lazy concurrent list-based set algorithm. In: Ball, T., Jones, R.B. (eds.) CAV 2006. LNCS, vol. 4144, pp. 475–488. Springer, Heidelberg (2006)
8. Harris, T.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J.L. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001)
9. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006)

10. Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
11. Herlihy, M., Wing, J.: Linearizability: A correctness condition for concurrent objects. ACM Trans. Program. Lang. Syst. 12(3), 463–492 (1990)
12. Holzmann, G.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley, Reading (2003)
13. Lea, D.: The java.util.concurrent synchronizer framework. In: CSJP (2004)
14. Liu, Y., Chen, W., Liu, Y., Sun, J.: Model checking linearizability via refinement. In: Cavalcanti, A., Dams, D.R. (eds.) FM 2009. LNCS, vol. 5850, pp. 321–337. Springer, Heidelberg (2009)
15. Michael, M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA, pp. 73–82 (2002)
16. Michael, M., Scott, M.: Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester (1995)
17. Neven, F., Schwentick, T., Vianu, V.: Finite state machines for strings over infinite alphabets. ACM Trans. Comput. Logic 5(3), 403–435 (2004)
18. Segalov, M., Lev-Ami, T., Manevich, R., Ramalingam, G., Sagiv, M.: Abstract transformers for thread correlation analysis. In: Hu, Z. (ed.) APLAS 2009. LNCS, vol. 5904, pp. 30–46. Springer, Heidelberg (2009)
19. Vafeiadis, V., Herlihy, M., Hoare, T., Shapiro, M.: Proving correctness of highly-concurrent linearisable objects. In: PPOPP, pp. 129–136 (2006)
20. Vechev, M., Yahav, E., Yorsh, G.: Experience with model checking linearizability. In: Păsăreanu, C.S. (ed.) SPIN. LNCS, vol. 5578, pp. 261–278. Springer, Heidelberg (2009)

Local Verification of Global Invariants in Concurrent Programs

Ernie Cohen[1], Michał Moskal[2], Wolfram Schulte[2], and Stephan Tobies[1]

[1] European Microsoft Innovation Center, Aachen
{ernie.cohen,stephan.tobies}@microsoft.com
[2] Microsoft Research, Redmond
{michal.moskal,schulte}@microsoft.com

Abstract. We describe a practical method for reasoning about realistic concurrent programs. Our method allows global two-state invariants that restrict update of shared state. We provide simple, sufficient conditions for checking those global invariants modularly. The method has been implemented in VCC[1], an automatic, sound, modular verifier for concurrent C programs. VCC has been used to verify functional correctness of tens of thousands of lines of Microsoft's Hyper-V virtualization platform[2] and of SYSGO's embedded real-time operating system PikeOS.

1 Introduction

Verifying functional correctness of complex, low-level, shared memory, highly concurrent programs (e.g., operating system kernels) requires intricate global invariants. To scale to large systems, checking these invariants should be modular; in particular, invariance checking should obey program abstraction boundaries, so that e.g., low-level code can be checked without having to consider high-level invariants, and private low-level invariants can be changed without breaking high-level clients. However, the need for modularity conflicts with the practical need to have invariants that span multiple objects[3]; for example, high-level objects often have invariants that mention (public) fields of low-level objects.

One common approach to this problem is to impose on programs a structural discipline that syntactically restricts the form of object invariants. Such disciplines tend to work well for certain classes of programs but not for others, leading to a stream of extensions (e.g. the extension of the Spec# discipline [3,13] with friends [1], or freezing [16]). Another popular approach is based on separation logic [18] and its combination with rely-guarantee reasoning [11,19,10]. However, they too make methodological commitments that work well for certain classes of programs but less well for others (e.g., the choice to use fractional permissions

[1] VCC is available in source for academic use at http://vcc.codeplex.com/

[2] The Hypervisor verification is part of the Verisoft XT project supported by BMBF under grant 01IS07008.

[3] Objects mean collections of closely related data, e.g., regions of memory interpreted as structs in C.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 480–494, 2010.

rather than counting permissions [5]). Moreover, resource logics like separation logic have inherently higher complexity than ordinary state-based logics [6], and automation for separation logics (in their full generality) is far less developed than for conventional logics.

We propose a different approach that achieves modularity without going outside of ordinary logic: *locally checked invariants* (LCI). LCI assigns to each object (including threads) a two-state invariant, i.e., a predicate over pairs of states expected to hold for every pair of consecutive states in every execution.[4] Verification of a concurrent program reduces to checking that these invariants hold for every state update invoked by the program. Rather than syntactically restricting invariants, LCI imposes a *semantic* condition (i.e., a proof obligation) on the object invariants; this condition guarantees that *local* invariant checking (i.e., checking the invariants of objects actually changed in an update) suffices to prove preservation of all invariants. Because the restriction is semantic rather than syntactic, it allows for great deal of flexibility in structuring *global* invariants. In particular the kind of verification scaffolding that has to be built into other programming methodologies can be implemented syntactically in LCI at the program level, using ghost state and ordinary object invariants, allowing multiple methodologies to be used on a single program. As LCI is based on ordinary logic, it can be implemented on top of stock theorem provers.

LCI has been implemented in VCC [7], a verifier for concurrent C software that has verified the functional correctness of tens of thousands of lines of commercial concurrent C code. The use of LCI, rather than a specialized program logic, allowed VCC to be built on an established verification condition generator (Boogie [2]) and state-of-the-art theorem prover (Z3 [9]). This paper presents LCI on the example of a simple type-safe language, with a natural notion of objects. Applying LCI to C involved developing a type-safe, yet flexible, view of C memory, which is described separately [8]. Viewing states of execution, for which the invariants are evaluated, at the granularity of the hardware-atomic memory updates allowed us to verify fine-grained concurrent algorithms.

The main contributions of the paper are:

- the formulation of an *admissibility* condition for invariants, which permits the local checking of global invariants (Sect. 3),
- an encoding of common invariant disciplines as admissible invariants (Sect. 4),
- the introduction of *claims*, which are objects encapsulating stable, derived knowledge about system state, often necessary to verify concurrent algorithms (Sect. 5).

2 The Idea: Stable Invariants and Legal Actions

For simplicity of presentation, consider a state to be a heap mapping addresses to typed object values. By an *action*, we mean an ordered pair of states, representing

[4] States of execution refer to states possibly visible to other threads, e.g., in C every memory write yields a separate state. This allows for verification of fine-grained concurrent algorithms.

a transition from the first state (pre-state) to the second (post-state). Again for simplicity, we restrict consideration to actions that do not change the type of object stored at an address, i.e., the set of objects and their addresses are fixed (but this restriction is easily dropped).

Each object has a two-state invariant, determined by its type. Terms within object invariants wrapped with **old()** refer to the pre-state of a transition; terms not so wrapped refer to the post-state. An action is *safe* iff it satisfies the invariant of every object.

Here are two simple examples of object type definitions:

```
type Account {
  int val, creditLimit;
  inv(creditLimit ≤ val)
}
```

```
type Counter {
  int n;
  inv(n = old(n) ∨ n = old(n) + 2)
}
```

The invariants of these types can be read as follows: after a safe action, in every Account object, the value of the val field has to be greater or equal than the creditLimit field, and in every Counter object, the n field is unchanged or incremented by two. Note that any number of objects can be updated in a single action.

An object invariant can be interpreted on a single state by interpreting it over the *stuttering* action from that state, i.e., the action that goes from that state to itself. A state is *safe* if the stuttering action from that state is safe. We would like to be assured that actions always start from safe states, so we require program execution to start in a safe state, and require that all object invariants satisfy the following *reflexivity* property: if an action $\langle h_o, h \rangle$ satisfies the object invariant, then the stuttering action $\langle h, h \rangle$ also satisfies the invariant. If all object invariants are reflexive, then every safe action has a safe post-state. Invariants of both objects above are reflexive, whereas an invariant n > **old**(n) for the Counter, which requires an increment during each safe transition, would not be reflexive.

The invariants above refer only to fields of their object. If all invariants were like that, checking safety of an action (starting from a safe state) would require only checking the invariants of those objects updated by the action. But some invariants have to span multiple objects. Consider for example the following types, where % is the modulo operator and fields of object type are object references (like in Java or C#).

```
type ParityReading {
  Counter cnt;
  int parity;
  inv(cnt.n % 2 = parity)
}
```

```
type Low {
  Counter cnt;
  int floor;
  inv(floor ≤ cnt.n)
}
```

```
type High {
  Counter cnt;
  int ceiling;
  inv(cnt.n ≤ ceiling)
}
```

An action that updates x.cnt.n, where the type of x is any of the types defined above, might break the invariant of x. However, if the action preserves the invariant of x.cnt (i.e., increments x.cnt.n by two), then it cannot break the invariant of an x of type ParityReading or Low, but it can break the invariant of an x of type High. We therefore reject the definition of the type High as *inadmissible*, whereas

$$
\begin{array}{llll}
 & & \mathsf{type} & : \mathbb{Z} \to \mathbb{T} \\
\text{fields} & \mathbb{F} \equiv \{\mathsf{f}_0, \mathsf{f}_1, \dots\} & \mathsf{inv}_\tau & : \mathbb{H} \times \mathbb{H} \times \mathbb{Z} \to \mathbb{B} \quad \text{for } \tau \in \mathbb{T} \\
\text{types} & \mathbb{T} \equiv \{\mathsf{t}_0, \mathsf{t}_1, \dots\} & \mathsf{inv}(h_o, h, p) & \equiv \mathsf{inv}_{\mathsf{type}(p)}(h_o, h, p) \\
\text{integers} & \mathbb{Z} \equiv \{0, 1, -1, \dots\} & \mathsf{inv}_1(h, p) & \equiv \mathsf{inv}(h, h, p) \\
\text{heaps} & \mathbb{H} \equiv \mathbb{Z} \to (\mathbb{F} \to \mathbb{Z}) & \mathsf{legal}(h_o, h) & \equiv \mathsf{safe}_1(h_o) \Rightarrow \forall p.\ h_o[p] = h[p] \vee \mathsf{inv}(h_o, h, p) \\
\text{Booleans} & \mathbb{B} \equiv \{\textbf{true}, \textbf{false}\} & \mathsf{safe}(h_o, h) & \equiv \forall p.\ \mathsf{inv}(h_o, h, p) \\
 & & \mathsf{safe}_1(h) & \equiv \forall p.\ \mathsf{inv}_1(h, p)
\end{array}
$$

$$\mathsf{stable}(\tau) \equiv \forall p, h_o, h.\ \mathsf{type}(p) = \tau \wedge \mathsf{safe}_1(h_o) \wedge \mathsf{legal}(h_o, h) \Rightarrow \mathsf{inv}_\tau(h_o, h, p)$$

$$\mathsf{refl}(\tau) \equiv \forall p, h_o, h.\ \mathsf{type}(p) = \tau \wedge \mathsf{inv}_\tau(h_o, h, p) \Rightarrow \mathsf{inv}_\tau(h, h, p)$$

$$\mathsf{adm}(\tau) \equiv \mathsf{stable}(\tau) \wedge \mathsf{refl}(\tau)$$

Fig. 1. Definitions

it will allow the other type definitions. Note that the admissibility of a type can depend on the definitions of other types; e.g., ParityReading is only admissible because increments of Counter are always by two.

The essence of LCI is captured in the following (still informal) definitions. An action is *legal* iff it preserves the invariants of updated objects. A *stable* invariant is one that cannot be broken by legal actions (i.e., holds over legal actions). An *admissible* invariant is one that is stable and reflexive. It follows (by Theorem 1 below) that if all invariants are admissible, then every legal action from a safe pre-state is safe and has a safe post-state. Our methodology involves proving the admissibility of each invariant, and then proving that each action produced by the program is legal; this lets us conclude that all actions produced by the program are safe.

3 Formalization

We now formalize the insight from the last section (Fig. 1). Heaps $\mathbb{H}$ map integers (i.e., addresses) to objects, which are maps from field names to integers. The invariant function $\mathsf{inv}(h_o, h, p)$ returns true iff the action changing the state from h_o to h satisfies the invariant of (the object referenced by) p. For simplicity, the type of an object at a given address (given by the type function) is fixed. The inv function is constructed from type-specific invariants (inv_τ). E.g., the Counter's invariant $\mathsf{n} = \textbf{old}(\mathsf{n}) \vee \mathsf{n} = \textbf{old}(\mathsf{n}) + 2$ translates to:

$$\mathsf{inv}_{\mathsf{Counter}}(h_o, h, p) \equiv h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee h[p][\mathsf{n}] = h_o[p][\mathsf{n}] + 2$$

Our goal is to conclude the safety of an action while only checking its legality.

We first note that if all invariants are reflexive, then safe actions result in safe states:

Lemma 1. $(\forall \tau.\ \mathsf{refl}(\tau)) \Rightarrow \mathsf{safe}(h_o, h) \Rightarrow \mathsf{safe}_1(h)$

Stability is defined for type τ by $\mathsf{stable}(\tau)$ (in Fig. 1). Given a legal action ($\mathsf{legal}(h_o, h)$) that starts from a safe state ($\mathsf{safe}_1(h_o)$), we want to be able to

conclude that the action is safe, and thus (by Lemma 1) that the post-state is safe. By syntactic transformations of the definition of stable we get:

Lemma 2. $(\forall\tau.\ \mathsf{stable}(\tau)) \Rightarrow (\mathsf{safe}_1(h_o) \wedge \mathsf{legal}(h_o, h) \Rightarrow \mathsf{safe}(h_o, h))$

We call an invariant of type τ admissible ($\mathsf{adm}(\tau)$) if its reflexive and stable. This takes us to our main soundness theorem:

Theorem 1. *Let all types be admissible. Then, for a sequence of heaps* $h_0, h_1, ...$*:*

$$\mathsf{safe}_1(h_0) \wedge (\forall i.\ \mathsf{legal}(h_i, h_{i+1})) \Rightarrow (\forall i.\ \mathsf{safe}_1(h_i) \wedge \mathsf{safe}(h_i, h_{i+1}))$$

Proof. By combination of Lemmas 1 and 2. □

Thus, any sequence of *legal* actions starting from a safe state is a sequence of *safe* actions. Therefore, the verification system can generate proof obligations for legality, reflexivity and stability and, by Theorem 1, deduce global correctness. Reflexivity and stability depend only on invariants of data referenced from the current invariant; legality depends only on invariants of objects that are updated. As a consequence all these conditions can be checked modularly.

The proof of Theorem 1 is trivial given the carefully chosen definition of admissibility. Yet, in the following, we see that this notion is not overly restrictive and that many interesting invariants can naturally be made admissible.

Dependents and Fix-Points

We have mentioned that the invariant of High is inadmissible. However, if we actually wanted the Counter to have an external object restricting its changes, we could have done so by explicitly referring to the invariant of the High bound from the invariant of the Counter (multiple **inv**(...) clauses are syntactic sugar for their conjunction).

```
type Counter2 {
  int n;
  object b;
  inv(n = old(n) ∨ n = old(n) + 2)
  inv(b = old(b))
  inv(n = old(n) ∨ inv(b))
}
```

```
type High2 {
  Counter2 c;
  int ceiling;

  inv(c.b = this)
  inv(c.n ≤ ceiling)
}
```

The invariant of High2 is now admissible thanks to the back-reference b in the invariant of the Counter2. Any legal action touching n in a counter has to preserve invariant of its attached object b. The first-order translation of the Counter2's invariant that includes the reference to b's invariant is:

$$\begin{aligned}\mathsf{inv}_{\mathsf{Counter2}}(h_o, h, p) \ \equiv\ & (h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee h[p][\mathsf{n}] = h_o[p][\mathsf{n}] + 2) \wedge \\ & h[p][\mathsf{b}] = h_o[p][\mathsf{b}] \wedge (h[p][\mathsf{n}] = h_o[p][\mathsf{n}] \vee \mathsf{inv}(h_o, h, h[p][\mathsf{b}]))\end{aligned}$$

The definition of $\mathsf{inv}(\dots)$ has become recursive. We thus define the function $\mathsf{inv}(\dots)$ as any fix-point solution of the set of type-invariant-derived equations

like the one above. To ensure existence of a fix-point solution, we restrict the use of **inv**(...) in invariants to positive polarities.[5]

4 Structuring Invariants

LCI can be viewed as a "low-level" verification mechanism: admissibility, being a semantic check, allows for making arbitrary tradeoffs as to what to check where. In particular, any invariant can be made admissible by referencing it from the invariants of objects that it depends on (as in the High2 example). This flexibility allows for encoding more sophisticated methodologies on top of LCI at the syntactic level, without jeopardizing soundness. Indeed, LCI makes it easy to soundly add (and mix) different formal methodologies within a single program.

Central to the encoding of methodologies on top of LCI is the use of *ghost state* — state added to the program to aid reasoning. An implementation should check that ghost code (code that references ghost state) does not update non-ghost state and is total (terminating); these conditions guarantee that the ghost code and state can be erased without effecting behavior of the program on non-ghost state. Ghost code is not only used to encoding methodologies; it also provides a critical tool for program annotation, allowing the programmer to communicate to the verifier important intuitions as to why the program works. For example, ghost objects can be used to encode rights and knowledge, and ghost code can be used to create, destroy, and transfer these objects during program execution.

Here, we describe how some of the methodologies used in VCC are encoded syntactically on top of LCI (summarized in Fig. 2). We apply a syntactic transformation to every user-defined type, which adds additional fields and invariants, and which also weakens the user-defined invariant ψ. For every instance of an user-defined type, we also add an instance of the helper OwnerCtrl type. We first describe disabling invariants (Sect. 4.1) and ownership trees (Sect. 4.2); these provide the Concurrent Spec# [13] ownership system. We then describe handles (Sect. 4.3), a generalized form of read permissions [5].

4.1 Validity

Theorem 1 assumes that the initial state is safe. However, most languages allow for dynamic creation (and perhaps disposal) of objects, and object's invariants usually do not hold prior to completion of initialization nor after the start of destruction. Additionally, when objects are only accessed sequentially it is convenient to temporarily disable the invariant, perform several updates on an object, and re-enable its invariant. For those reasons we introduce a valid field, defined

[5] Let L be the power set lattice of $\mathbb{H} \times \mathbb{H} \times \mathbb{Z}$, let $f : L \to L$ be a function such that $\langle h_o, h, p\rangle \in f(I)$ if and only if $\mathsf{inv}_{\mathsf{type}(p)}(h_o, h, p)$ with inv replaced with characteristic function of I. Because $\mathsf{inv}(\dots)$ only occurs positively in $\mathsf{inv}_{\mathsf{type}(p)}(\dots)$ f is monotonic, and therefore by Knaster-Tarski theorem f has a fix-point, which we use as inv. An unconditional cycle in inv definitions might yield an interpretation of **false**, which is sound but will prevent successful verification.

```
1  type τ {
2    F
3
4    // Validity, Sect. 4.1
5    ghost bool valid;
6    inv((old(valid) ∨ valid) ⇒ ψ)
7
8    // Ownership, Sect. 4.2
9    ghost OwnerCtrl ctrl;
10   inv(unchg(ctrl))
11   inv(ctrl.subject = this)
12   inv(unchg(valid) ∨ inv(ctrl))
13   // for every f ∈ F
14   inv(¬valid ⇒
15     unchg(f) ∨ inv(ctrl.owner))
16 }
```

```
17 ghost type OwnerCtrl {
18   object owner, subject;
19   inv(unchg(subject))
20   inv(unchg(owner) ∨ inv(owner))
21   inv(unchg(owner) ∨ inv(old(owner)))
22   inv(unchg(subject.valid) ∨ inv(owner))
23   inv(type(owner) = Thread ∨ subject.valid)
24   inv(subject.ctrl = this)
25   // Handles, Sect. 4.3
26   set<Handle> handles;
27   inv(unchg(handles) ∨ inv(owner))
28   inv(∀(Handle h;
29         h ∈ old(handles) ∧ h ∉ handles
30           ⇒ ¬h.valid))
31   inv(handles = {} ∨ subject.valid)
32 }
```

Fig. 2. Every user (i.e., not OwnerCtrl) type τ with fields $\mathcal{F}$ is transformed by adding implicit ghost fields and invariants. The user invariant, ψ, is weakened. Additionally an ownership control type is added (but not transformed). **unchg**(f) is short for **old**(f)= f.

for every object type. Line 6 of Fig. 2 says that user invariants have to hold only when the object is valid in the pre- and/or the post-state. In particular, single-state user-invariants only have to hold when the object is valid. We define the valid field to be **false** in the initial state, so the initial state is safe. An object is typically made valid at the end of initialization and invalid before its destruction.

4.2 Ownership

How does the introduction of validity effect the admissibility of invariants? Consider, for example, the invariant of an object low of type Low. To make use of the "counter only goes up" property of low.cnt, low now needs to know that low.cnt is valid. In a language with explicit destruction, this can be rephrased as: what prevents somebody from destroying low.cnt? One possible solution is to designate low as the *owner* of low.cnt; informally, we can think of this meaning that low.cnt is "part of" low.

Formally, ownership is expressed by adding a ghost ctrl field to every object and making it point to an instance of ownership control type (again Fig. 2). Now the field x.ctrl.owner points to the object that owns x. This encoding allows the ownership relation to dynamically change during execution, but guarantees that each object always has a unique owner.

The invariants governing OwnerCtrl say that it always refers to the same subject (line 19), ownership transfer is allowed, but requires a check of the invariants of the old and new owner (line 20 and line 21), and only threads can own invalid objects (line 23; thread invariants will be discussed in Sect. 5.1). Note that the use of a separate ownership control object means that the invariants of the object itself do not have to be checked when its owner changes. Note also that these

```
type Lock {
  bool locked;
  ghost object rsc;
  inv(unchg(rsc))
  inv(¬locked ⇒
    rsc.ctrl.owner = this)
}
```

```
void Acquire(Lock l, ghost Handle h)
  requires(h.ctrl.owner = me ∧ h.valid ∧ h.obj = l)
  ensures(l.rsc.ctrl.owner = me)
{
  do ⟨ prev := l.locked;
       if (¬prev) {
         l.locked := true;
         ghost { l.rsc.ctrl.owner := me; } } ⟩
  while (prev);
}
```

Fig. 3. A spin-lock. The semantics of actions (Acquire) is explained in Sect. 5.

invariants allow ownership cycles, but in practice the ownership relation defines a forest, except that the tree roots (threads) own themselves.

We could now make Low own and thus control validity of its Counter by adding the user invariant cnt.ctrl.owner = **this**, which entails cnt.valid. Such an invariant is admissible: the additional invariant on Counter from line 10 ensures that any legal action leaves cnt.ctrl unchanged, and an attempt to update cnt.ctrl.owner needs to conform with both the old and new owner's invariants (see lines 20 and 21). Thus, Low cannot loose control of its Counter.

Conversely, an invariant of the form **type**(ctrl.owner)= Low in Counter would not be admissible: the control type enforces the invariant of owners, not subjects, so changing the owner of a subject (which only involves changing the owner field in the OwnerCtrl type and not the subject itself) could break the invariant. This is intentional: synchronization mechanisms, like *spin-locks*, should be polymorphic in the type of the data that they protect, but the implementation of these mechanisms needs to manipulate ownership of the protected objects. For example, acquiring a lock transfers ownership of the lock-protected resource (rsc in Fig. 3) to the calling thread. Similarly, generic containers (e.g. lists, stacks) should be polymorphic in the type of the objects they hold.

4.3 Handles

Ownership does not provide for sharing; at most one thread can (transitively) own an object at any time, and only that thread can deduce (through ownership) that the object is valid. But sharing is often necessary. For example, the whole point of a spin-lock is that multiple threads can try to acquire it simultaneously; all of these threads must know that the lock is valid.

One way to provide sharing is with *handles*. Consider the following type (on which we apply the transformation from Fig. 2):

```
ghost type Handle {
  object obj;
  inv(unchg(obj) ∧ this in obj.ctrl.handles ∧ obj.valid)
}
```

The invariant of Handle is admissible, because the invariants of OwnerCtrl prevent removal of a valid handle from the handles set (line 30) and prevent the object from being invalidated as long as there are outstanding handles (line 31). Multiple clients can each own valid handles on the shared object, and each of these handles guarantees validity of the shared object.

When multiple clients rely on an object, the object's owner typically keeps track of the clients, e.g., by maintaining a reference counter. An example of such scenario is a *reader-writer lock*. Acquiring a read lock returns to the caller a handle on the resource. Acquiring a writer lock waits for the reader count (correlated in the invariant with the cardinality of the handle set) to drop to zero and transfers the ownership of the resource from the lock to the acquiring thread. This thread can then invalidate the resource and update its fields.

A handle allows the fields of obj to be changed (subject to checking obj's invariant); the validity of obj is needed in the case that obj is owned by another thread. This allows threads to race in a controlled manner, for example by trying to acquire a lock. A handle on obj can also be viewed as a read permission on the fields f of obj for which obj.valid $\wedge$ **inv**(obj)$\Rightarrow$ **unchg**(obj.f): as long as the handle exists, these fields cannot change. This is a counting, rather than fractional, permission[6]; but unlike permissions in separation logic, handles are first-class objects, so one can create handles on handles with similar results to splitting fractional permissions (cf. [5]).

As a more concrete example consider the following modification of the type Low:

```
type Low2 {
  Counter cnt;
  int floor;
  inv(floor ≤ cnt.n)
  ghost Handle cntH;
  inv(cntH.ctrl.owner = this ∧ cntH.obj = cnt)
}
```

Adding the handle cntH on cnt makes the invariant of the Low2 admissible, while being able to share Counter objects between clients. A similar modification needs to be made to ParityReading and High2.

5 Verifying Actions

LCI treats procedures as sequences of atomic actions performed by a single thread (denoted by **me**), with possible interference of safe actions of other threads in between. To verify a procedure, one needs to check legality of all its actions, assuming only safety of interfering actions (that is without looking at the code of other procedures). For example, VCC achieves that by translating the sequence of actions into a sequential Boogie program, and verifying it using weakest preconditions calculus [2]. After translation of each action, we assert its legality

[6] Fractional permissions can also be implemented on top of LCI, but we have not found them necessary.

and assign an arbitrary value to the entire heap. We assume this new heap to represent a safe state (per Theorem 1) and that it was constructed from the old one by zero or more safe actions of other threads (which has consequences to thread-local data, see below). As an example, consider the following code, using $\langle \ldots \rangle$ to denote atomic actions:

```
1  void incr(Counter c) {
2    ⟨ a := c.n; ⟩
3    ⟨ if (c.n = a) c.n := a + 2; ⟩
4    ⟨ b := c.n; assert(a < b); ⟩
5  }
```

We first read the value of the counter into a local variable a. Then, using an atomic compare-and-swap operation (in-lined here for clarity), we increment the value of the counter, provided it did not change in between. Finally, we read the new value of the counter into b and want to statically prove $a < b$. This should be provable because the invariant of Counter ensures that the counter can only grow, and so either the compare-and-swap succeeds in storing the incremented value, or it fails because somebody else incremented it between lines 2 and 3.

We verify incr as follows. Suppose for a moment that we know c remains valid throughout execution of incr. We check the legality of the first action; because it modifies only local variables, this is trivial. Afterwards, we simulate arbitrary, but safe, interference by other actions by assigning an arbitrary safe value to the heap. We then check legality of the second action performed on that new heap and so on. Local variables, remain unchanged between actions of the current thread. Because locals do not change, and c.n can only grow, from line 2, we have $a \leq c.n$ and after line 3 even $a < c.n$. Thus, the assertion does indeed hold.

This kind of reasoning makes use of the fact that any property ψ of data on the heap is preserved, provided that it is maintained by safe actions of other threads, i.e., $\forall h_o, h.\ \mathsf{safe}(h_o, h) \wedge h_o[\mathbf{me}][\mathsf{state}] = h[\mathbf{me}][\mathsf{state}] \wedge \psi(h_o) \Rightarrow \psi(h)$, where $h[\mathbf{me}][\mathsf{state}]$ is used to encode the local state of the current thread. However, we cannot expect all such ψ's to be automatically inferred by the theorem prover as needed. In the following, we deal with situations where the inference of a suitable ψ is beyond the capabilities of the theorem prover. Preservation of some common properties, like behavior of thread-local data, can be proven as lemmas once and for all (see Sect. 5.1) and then be built into the verifier. Other properties need to be taken care of by the user (Sect. 5.2), using LCI's existing machinery.

5.1 Thread-Local Data

We now come back to the fact that we need validity of a Counter c throughout the execution of incr(c): if some other thread would invalidate c while incr(c) is executing, c.n could potentially decrease and the assertion from line 4 can no longer be proven. Simply making the invariant of some other object o guarantee c.valid, does not help, as it would only defer the validity of c to the validity of o. The natural place to start validity deduction is **me**, i.e., the Thread object representing the current thread of execution.

The field o.f is *thread-local* data of thread t iff the invariant of t can admissibly prevent any other threads from changing it, e.g., by including a formula like **unchg**(**this**.state) $\wedge\ \phi \Rightarrow$ **unchg**(o.f), where ϕ encodes additional conditions, like o.ctrl.owner = **this**. The thread invariants are defined to be conjunctions of all such admissible formulas. Thus, they prevent any interference they possibly can. We approximate that by assuming **inv**(**me**) to be true and leaving other thread invariants unspecified when verifying actions. Hence it is sound to assume that thread-local data does not change between actions of the current thread.

In particular an object invariant **unchg**(f) $\vee$ **inv**(ctrl.owner) makes the field f local data of the owning thread (provided that ctrl.owner is a thread). Thus the valid and owner fields are both local to the owning thread. Additionally, per line 15 of Fig. 2, all fields of invalid objects are local to the owning thread, too.

As a consequence, if **me** owned c, that would be enough to verify incr(c): the fields c.ctrl.owner and c.valid would be thread-local. But clearly, incr(c) is designed so that multiple threads can execute it concurrently on the same c. So instead we reuse the Handle type: **me** owns a (initially valid) handle h, therefore h.ctrl.owner, h.valid, and h.obj are thread-local, and **inv**(h) implies c.valid.

5.2 Claims

Verification of incr(), in addition to a handle, relied on a lemma that the assertion a $\leq$ c.n is preserved by legal updates. To persist such a property ψ between LCI actions one can use a *claim* — an object with invariant ψ. The stability of the invariant (checked by an LCI verifier) implies the lemma. In case of incr() the existing type Low2 can be used as a claim. A claim is ghost, as it is only a verification device, and so we can insert updates of its multiple fields in the middle of the physical atomic actions. In the code below, we pass a pre-allocated, thread-local Low2 claim object and, for simplicity, ignore its creation. We also add an invariant making fields of Low2 thread-local.

```
type Low2 { // ...
  inv((unchg(floor) ∧ unchg(cntH) ∧ unchg(cnt)) ∨ inv(ctrl.owner))
}
void incr(Counter c, ghost Handle h, ghost Low2 cl)
  requires(h.obj = c ∧ h.ctrl.owner = me ∧ h.valid)
  requires(¬cl.valid ∧ cl.ctrl.owner = me)
{
  ⟨ a := c.n ; ghost { cl.cnt = c; cl.cntH = h; h.ctrl.owner := cl;
                       cl.floor := a; cl.valid := true; } ⟩
  ⟨ if (c.n = a) { c.n := a + 2; }
    ghost { cl.floor := a + 1; } ⟩
  ⟨ b := c.n; assert(a < b); ⟩
}
```

After the first read, we initialize the Low2 claim cl with a as a lower bound for c according to its invariant. The next action first tries to increment the counter, and regardless of the result increments the floor field of the claim. This is a legal action: if we incremented the counter, then a + 2 is a new lower bound. If we did

not, $\mathsf{c.n} \neq \mathsf{a}$ must hold and Low2's invariant (which holds just before the action) additionally entails $\mathsf{c.n} \geq \mathsf{a}$, and thus $\mathsf{a} + 1$ is a lower bound for the counter. Correctness of last action simply follows from cl's invariant as we have made fields of the claim thread-local by requiring changes to satisfy **inv(me)**. Thus, a claim is used to derive a inductive property of an invariant.

Claims are often needed when verifying programs using LCI. For this reason, VCC provides a syntax for creating a new claim type and instance, along with the required handles, in a single statement. The claim object also captures the values of locals at the point where it is created. Using this syntactic sugar, the first action in our program could be written without introduction of the type Low2 as:

$\langle\ \mathsf{a} := \mathsf{c.n};\ \textbf{ghost}\ \{\ \mathsf{s} := \textbf{claim}(\mathsf{h},\ \mathsf{c.n} \geq \mathsf{a})\ \}\ \rangle$

VCC claims are still first-class objects, so it is possible to own them, pass them around, and store them in other objects. In particular, it allows for proving a lemma in one context and using it in another.

6 Evaluation and Related Work

The LCI verification methodology was developed for and along with the VCC verification tool. Our goal for VCC was the sound verification of functional correctness of industrial-strength concurrent C code, driven by program-level annotations on the code base. Because we were after a scalable, industrial verification process driven by software engineers (rather than verification engineers), we avoided interactive proof checking and logics that are difficult to automate efficiently (e.g., higher-order logics and separation logics). Ghost data and code was used to workaround limitations of first-order logic, e.g., the reachability relation was expressed with a ghost field holding the set of reachable nodes, which needed to be updated explicitly, using ghost code. We believe that developers are well equipped to write substantial amounts of code manipulating ghost state (to manipulate ownership, claims, as well as abstractions like the reachability above), and we know that such techniques scale to complex problems.

The driving application for VCC development was the verification of the Microsoft Hyper-V hypervisor. The hypervisor (about 100K lines of C code) sits directly on multi-processor x64 hardware, and provides a number of virtual multi-processor x64 machines (with some additional instructions). The hypervisor is highly optimized for multi-core hardware, so in addition to typical OS components (e.g., scheduler and memory allocator) it contains a number of custom concurrency control mechanisms and algorithms, mostly using fine-grained concurrency control. For the last two years, the Hyper-V verification project has focused on annotating the existing code base with invariants and function contracts and checking these annotations, to prove that the Hyper-V simulates (a model of) the actual x64 hardware. So far, about 1/3 of the code base has been annotated. VCC is also used to verify PikeOS, an embedded real-time OS [4].

LCI has proven to be powerful enough to express all required specifications, ranging from low-level details, like concurrency primitives, to higher-level abstractions like virtual machine partitions. LCI was relatively easy to implement in VCC; however specifications need to be carefully formulated to stay off of unfruitful, non-terminating, or very time-consuming proof paths. Still, it was possible to massage the specifications to the point where the Z3 theorem prover was accepting them with certain robustness (i.e., small changes in specifications would not make it fail).

The expressivity of LCI is beyond that of other verification tools and there are no commonly accepted benchmarks in this area, so we cannot easily compare run-times. To give you an idea of the performance that we achieve, here are the times consumed by the verification of the Hyper-V sources on a single 2.33 GHz Intel Xeon core:

	# of checks	min	max	avg	median
admissibility	152	0.5s	50s	15s	13.6s
functions	367	0.4s	2581s	50.5s	12.8s

Actual turnaround times on our 8-core machine (multiple admissibility checks and function verifications can be parallelized trivially) is well below 2 hours. For most problems, a memory limit of 200MB suffices, while a handful of problems require more memory but never exceeding 1GB.

The typical VCC work flow starts by running VCC on the initial version of the code and specification. In case of a verification error either the specification or the code needs to be fixed, the decision is up to the VCC's user, who is aided by a model viewer, showing the counter-example found by the theorem-prover. The user then runs VCC again and the process repeats, typically multiple times. From this point of view the most important performance characteristic is the time it takes to verify a single function.

Many of the ideas of VCC are rooted in the Spec# project [3]. However, unlike Spec#, VCC is based on a tiny core, namely LCI, allowing users to extend the methodology at will. In fact, Spec#'s sequential methodology, comprising reps, peers, visibility, and observers, can be expressed in LCI. Also our thread-ownership discipline follows Concurrent Spec# [13]. The difference is that in LCI one can also verify lock *implementations* (see Fig. 3 and [12]), instead of treating locks as primitives. Finally, VCC also adopted Spec#'s framing mechanism: procedures list the objects' ownership domains that they are allowed to write.

An alternative promising approach to addressing the modular verification problem of concurrent programs is taken by the separation logic [18] community. Concurrent separation logic [17] divides the heap into a thread-local and a shared part. The shared part is governed by a single-state invariant. Extensions like SAGL [11] and RGSep [19] use two-state rely/guarantee predicates for the shared part. Finally, LRG [10] introduces the separating conjunction over two-state predicates. All these approaches require specialized resource logics, whereas our goal has been to stick to first-order logic to be able to leverage existing high-performance automated theorem provers.

The Chalice program verifier [15] also uses a resource logic, but it reasons about permissions at the syntactic level, and pushes the remaining proof obligations to the theorem prover. While restricted at the permission level, the specifications can be stronger in their classical parts.

A current disadvantage of LCI is its high annotation overhead: Spec#, some separation logic tools, and Chalice all have much lower overhead in cases they are expressive enough. Part of this is intentional; we intended to provide a powerful "low-level" program verifier, and to later introduce syntactic sugar as we learned what kinds of abstractions and abbreviations were most effective. In VCC, which works at slightly higher level of abstraction than the bare-bone LCI presented in this paper, we have found it to be in the order of one line of annotation per line of code. As a comparison, the impressive seL4 verification project [14], where similar properties were verified (but in a much smaller code base of sequential code), is reporting 2:1 overhead for specifications and 10:1 overhead for the proofs (in LCI the proofs are automatic, modulo the aforementioned massaging).

7 Conclusion

LCI is a modular verification methodology for concurrent programs. It introduces semantic admissibility condition on two-state invariants, which guarantees that updates can be checked locally. LCI has been proven to scale in practice and be expressive enough for industrial program verification.

Acknowledgements. Thanks to VCC users and people who provided the infrastructure: Artem Alekhin, Eyad Alkassar, Mike Barnett, Nikolaj Bjørner, Holger Blasum, Sebastian Bogan, Sascha Böhme, Matko Botinĉan, Vladimir Boyarinov, Markus Dahlweid, Ulan Degenbaev, Lieven Desmet, Sebastian Fillinger, Mark A. Hillebrand, Tom In der Rieden, Bruno Langenstein, Dirk Leinenbach, K. Rustan M. Leino, Wolfgang Manousek, Stefan Maus, Leonardo de Moura, Andreas Nonnengart, Steven Obua, Wolfgang Paul, Hristo Pentchev, Elena Petrova, Thomas Santen, Norbert Schirmer, Sabine Schmaltz, Peter-Michael Seidel, Andrey Shadrin, Alexandra Tsyban, Sergey Tverdyshev, Herman Venter, and Burkhart Wolff.

References

1. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: Kozenand, D., Shankland, C. (eds.) MPC 2004. LNCS, vol. 3125, pp. 54–84. Springer, Heidelberg (2004)
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. Barnett, M., DeLine, R., Fähndrich, M., Leino, K.R.M., Schulte, W.: Verification of object-oriented programs with invariants. Journal of Object Technology 3(6), 27–56 (2004)

4. Baumann, C., Beckert, B., Blasum, H., Bormer, T.: Better avionics software reliability by code verification – A glance at code verification methodology in the Verisoft XT project. In: Embedded World 2009 Conference, Nuremberg, Germany, March 2009. Franzis Verlag (to appear, 2009)
5. Bornat, R., Calcagno, C., O'Hearn, P., Parkinson, M.: Permission accounting in separation logic. SIGPLAN Not. 40(1), 259–270 (2005)
6. Calcagno, C., Yang, H., O'Hearn, P.W.: Computability and complexity results for a spatial assertion language for data structures. In: APLAS, pp. 289–300 (2001)
7. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: Urban, C. (ed.) TPHOLs 2009. LNCS, vol. 5674, pp. 23–42. Springer, Heidelberg (2009) (invited paper)
8. Cohen, E., Moskal, M., Schulte, W., Tobies, S.: A precise yet efficient memory model for C. In: Workshop on Systems Software Verification (SSV 2009). Electr. Notes Theor. Comput. Sci., vol. 254, pp. 85–103. Elsevier Science B.V., Amsterdam (2009)
9. de Moura, L., Bjorner, N.: Z3: An Efficient SMT Solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
10. Feng, X.: Local rely-guarantee reasoning. In: Shao, Z., Pierce, B.C. (eds.) POPL, pp. 315–327. ACM, New York (2009)
11. Feng, X., Ferreira, R., Shao, Z.: On the relationship between concurrent separation logic and assume-guarantee reasoning. In: De Nicola, R. (ed.) ESOP 2007. LNCS, vol. 4421, pp. 173–188. Springer, Heidelberg (2007)
12. Hillebrand, M.A., Leinenbach, D.C.: Formal verification of a reader-writer lock implementation in C. In: Workshop on Systems Software Verification (SSV 2009). Electr. Notes Theor. Comput. Sci., vol. 254, pp. 123–141. Elsevier Science B.V., Amsterdam (2009)
13. Jacobs, B., Smans, J., Piessens, F., Schulte, W.: A simple sequential reasoning approach for sound modular verification of mainstream multithreaded programs. Electr. Notes Theor. Comput. Sci. 174(9), 23–47 (2007)
14. 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: POPL, pp. 207–220. ACM, New York (2009)
15. Leino, K.R.M., Müller, P.: A basis for verifying multi-threaded programs. In: Castagna, G. (ed.) ESOP 2009. LNCS, vol. 5502, pp. 378–393. Springer, Heidelberg (2009)
16. Leino, K.R.M., Müller, P., Wallenburg, A.: Flexible immutability with frozen objects. In: Shankar, N., Woodcock, J. (eds.) VSTTE 2008. LNCS, vol. 5295, pp. 192–208. Springer, Heidelberg (2008)
17. O'Hearn, P.W.: Resources, concurrency, and local reasoning. Theor. Comput. Sci. 375(1-3), 271–307 (2007)
18. Reynolds, J.C.: Separation logic: A logic for shared mutable data structures. In: LICS, pp. 55–74. IEEE Computer Society, Los Alamitos (2002)
19. Vafeiadis, V., Parkinson, M. J.: A marriage of rely/Guarantee and separation logic. In: Caires, L., Vasconcelos, V.T. (eds.) CONCUR 2007. LNCS, vol. 4703, pp. 256–271. Springer, Heidelberg (2007)

Abstract Analysis of Symbolic Executions

Aws Albarghouthi[1], Arie Gurfinkel[2], Ou Wei[1,3], and Marsha Chechik[1]

[1] Department of Computer Science, University of Toronto, Canada
[2] Software Engineering Institute, Carnegie Mellon University, USA
[3] Nanjing University of Aeronautics and Astronautics, China

Abstract. Multicore technology has moved concurrent programming to the forefront of computer science. In this paper, we look at the problem of reasoning about concurrent systems with infinite data domains and non-deterministic input, and develop a method for verification and falsification of safety properties of such systems. Novel characteristics of this method are (a) constructing under-approximating models via symbolic execution with abstract matching and (b) proving safety using under-approximating models.

1 Introduction

Concurrency has moved to the forefront of computer science due to the fact that future speedups of software rely on exploiting concurrent executions on multiple processor cores. Thus, the problem of creating correct concurrent programs is now paramount. Reasoning about such programs, i.e., determining whether properties of interest hold or fail in them, has always been difficult, especially if we consider "realistic" programs with infinite data domains (i.e., integer variables) and non-deterministic input. An example of such a program is the simple two-process mutual exclusion protocol shown in Fig. 1, where integer variables x and y are set non-deterministically (see Section 2 for more detail).

Approaches to reason about concurrent systems can be split into four categories. (1) "Classical" model-checking techniques, e.g., [18], were created to enumerate all reachable states of the program. Such techniques provide both verification and falsification information and are very effective when the state-space of the program is finite. However, they do not scale well for programs with large state-spaces and do not apply to those with infinite state-spaces. (2) Techniques like [3,16,7] build an *over-approximation* of program behaviours, via static analysis. These techniques can handle large/infinite state-spaces, are effective for verification purposes, but are not particularly well suited for finding bugs. (3) Techniques like [24,23,4,21] explore an *under-approximation* of feasible program behaviours. These techniques are often inexpensive and very effective for finding bugs; they are, however, often unable to prove correctness of programs. (4) Recently, researchers have been exploring the combination of under- and over-approximation by combining dynamic and static analysis techniques, respectively. Examples of this approach include [23] and the YOGI project [22]. These techniques are effective both for verification and for falsification of safety

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 495–510, 2010.

Process 1
$t_1 :\ pc_1 = 1 \longrightarrow b := b + 1,\ pc_1 := 2$
$t_2 :\ pc_1 = 2 \ \wedge\ x \leq y \ \wedge\ b = 2 \longrightarrow pc_1 := 3$
$t_3 :\ pc_1 = 3 \longrightarrow x := nondet,\ pc_1 := 2$

Process 2
$t_4 :\ pc_2 = 1 \longrightarrow b := b + 1,\ pc_2 := 2$
$t_5 :\ pc_2 = 2 \ \wedge\ x > y \ \wedge\ b = 2 \longrightarrow pc_2 := 3$
$t_6 :\ pc_2 = 3 \longrightarrow y := nondet,\ pc_2 := 2$

Fig. 1. A simple two-process mutual exclusion protocol with inputs x and y

properties but, with the exception of [23], have been limited to sequential programs [26,15,19,13]. Our work fits into this category.

In this paper, we propose a novel approach for automatically checking safety properties of *reactive* concurrent programs (over a finite number of threads) with *non-deterministic input* and *infinite data domains*. Handling these features allows us to target programs with infinite state-spaces, uninitialized variables, and communication with an external environment (e.g., user interaction). Our approach combines symbolic execution (to deal with non-deterministic input) and predicate abstraction (to deal with infinite data domains) in an abstraction-refinement cycle. Symbolic exploration proceeds along a path until it discovers two symbolic states that match to the same abstract state – the process is called *abstract matching* [17]. It produces an under-approximating abstract model that is more precise, in terms of feasible program behaviours it captures, than under-approximation techniques based on *must* transitions [24], concrete model checking and abstract matching [23], and weak reachability [4]. Since we only explore feasible program behaviours, all errors we encounter are real. We then analyse the abstract model to determine if it is also an over-approximation of the reachable concrete program states. If so, we conclude safety; otherwise, we refine the abstraction, adding predicates not to remove spurious counterexamples (as in the CEGAR framework [8]) but to enable us to explore more feasible program behaviours. To our knowledge, this is the first software verification algorithm combining symbolic execution with predicate abstraction and refinement. Our contributions are thus as follows: *(i)* a novel method for improving precision of under-approximating models by constructing them via a combination of symbolic execution and abstract matching; *(ii)* a novel technique for proving safety using under-approximating models; *(iii)* an implementation based on [23] and an empirical evaluation comparing the two approaches.

The rest of this paper is organized as follows. In Section 2, we give a general overview of the approach, illustrating it on the example in Fig. 1. We define the notation and provide background for the remainder of the paper in Section 3. Section 4 presents our approach in more detail, and Section 5 describes our implementation and experimental results. Section 6 compares our approach with related work. We conclude in Section 7 with the summary of our contributions and suggestions for future work.

2 Overview

In this section, we illustrate our approach on a simple two-process mutex protocol shown in Fig. 1. The protocol is written in a simple guarded command language. Initially, variables x and y are undefined (i.e., they can have an arbitrary value),

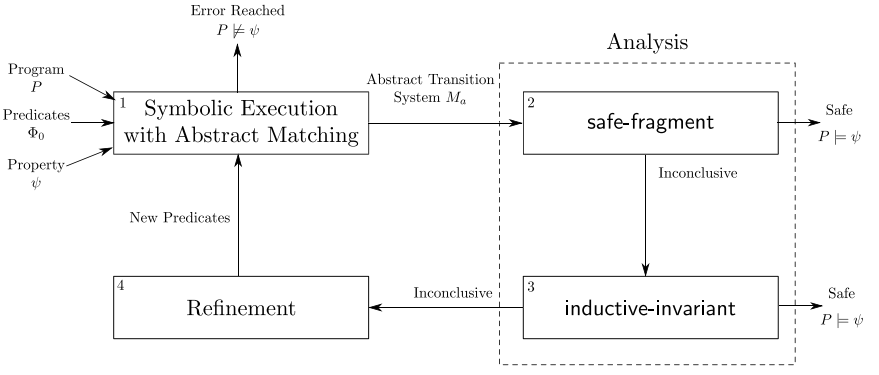


Fig. 2. Abstract analysis of symbolic executions

b is 0, pc_1 is 1, and pc_2 is 1. Process 1 starts at $pc_1 = 1$, increments b, and moves to $pc_1 = 2$ (transition t_1). At $pc_1 = 2$, it waits until b becomes 2 and x is less than or equal to y and proceeds to its critical section at $pc_1 = 3$ (transition t_2). At $pc_1 = 3$, it sets x non-deterministically (modelling input) and returns to $pc_1 = 2$ (transition t_3). Process 2 behaves analogously but uses process counter pc_2 and resets variable y in its critical section. We aim to show that this protocol satisfies the mutual exclusion property: a state where $pc_1 = 3 \wedge pc_2 = 3$ is not reachable.

The high-level overview of our approach is shown in Fig. 2. To determine whether a safety property ψ holds in a program P, we compute an abstract transition system, M_a, of P w.r.t. some initial set of predicates Φ_0 using symbolic execution with abstract matching. The state-space of M_a is an *under-approximation* of reachable abstract states of P. If an error is found during the symbolic execution step, we report P as unsafe and terminate. Otherwise, $M_a \models \psi$, and M_a is passed to the analysis phase which checks, via two separate steps, whether the state-space of M_a is also an *over-approximation* of P. If so, we are able to conclude that P is safe. Otherwise, we refine the set of predicates and repeat the entire process.

Our approach follows an abstraction-refinement loop, but differs from the standard CEGAR framework [8] in two ways: (1) we compute an *under-approximating abstraction* of P (using symbolic execution); (2) we do not rely on counterexamples to perform the refinement. In the rest of this section, we discuss each step of our approach in turn.

Symbolic Execution with Abstract Matching. Fig. 3(a) shows a symbolic execution tree of the program in Fig. 1. The initial set of predicates, $\Phi_0 = \{x \leq y, b = 2\}$, consists of all the predicates from the guards of the program. A *symbolic state* consists of the current values of variables conjoined with the *path condition* that has to be satisfied in order to reach this state. In Fig. 3(a), each state is represented as a box, with values of variables in the order (pc_1, pc_2, x, y, b) appearing in the top and the path condition – in the bottom. For example, state

s_1 is $(pc_1 = 1, pc_2 = 1, x = x_0, y = y_0, b = 0) \wedge (x_0 \leq y_0)$, where x_0 and y_0 are symbolic constants representing the initial value of x and y, respectively.

We use traditional symbolic execution with one additional constraint: in each symbolic state, each predicate from Φ_0 must be either satisfied or refuted. If necessary, we split a symbolic state by strengthening its path condition. For example, the initial state of the program in Fig. 1, $s_0 = (pc_1 = 1, pc_2 = 1, x = x_0, y = y_0, b = 0)$, neither satisfies nor refutes the predicate $x \leq y$. Thus, it is split into states s_1 and s_2 that satisfy and refute $x \leq y$, respectively. They become the new initial states. Similarly, states s_5 and s_6 are obtained by splitting a symbolic successor of s_4. Our constraint may increase the number of symbolic states, but it ensures that each symbolic state corresponds to (or matches with) a unique valuation of all of the predicates in Φ_0. We call such a valuation an *abstract state*, and define a function $\alpha(s)$ mapping a symbolic state s into an abstract state.

The symbolic execution proceeds along a path until it discovers two states s and s' that match the same abstract state a, i.e., $\alpha(s) = \alpha(s') = a$. For example, the symbolic path starting at s_1 and passing through s_3 is stopped at s_5. Following [23], we call this process *abstract matching*. Since the range of α is finite, symbolic execution with abstract matching is guaranteed to terminate. Of course, execution also aborts whenever it encounters an error state.

An abstract transition system M_a is obtained from the symbolic execution tree by adding a transition between two abstract states a and a' iff there is a transition between two states s and s' in the symbolic execution tree, and $\alpha(s) = a$ and $\alpha(s') = a'$. The abstract transition system M_a for the execution tree in Fig. 3(a) is shown in Fig. 3(b). In the figure, each state is a valuation to $(pc_1, pc_2, x \leq y, b = 2)$. For example, $\alpha(s_1) = a_1$ and $\alpha(s_2) = a_2$. An error state is unreachable in M_a, so it is passed to the analysis phase.

Analysis: safe-fragment. This check is based on a notion of an *exact* transition. A transition between two abstract states a and b is *exact* iff every concrete state corresponding to a can transition to a concrete state corresponding to b. For example, transition $a_4 \rightarrow a_5$ in M_a is exact (denoted by a solid line) whereas transition $a_1 \rightarrow a_3$ in M_a is inexact (denoted by a dotted line).

We say that a set of states Q, called a *fragment*, of an abstract transition system M_a is *exact* iff (a) there is no outgoing transition from Q to other states in M_a, and (b) all internal transitions within Q are exact. Intuitively, all executions from concrete states corresponding to an exact fragment Q are trapped in it. We say that an exact fragment Q is *safe* iff it does not contain error states, i.e, it approximates a part of the state-space of P that cannot reach an error.

safe-fragment determines whether all paths in M_a are eventually trapped in a safe exact fragment. This is reduced to checking whether the transitions inside and between all nontrivial strongly connected components of M_a are exact. If so, M_a is an over-approximation of P (see Section 4.2); therefore, none of the executions of P can reach error and thus P is safe.

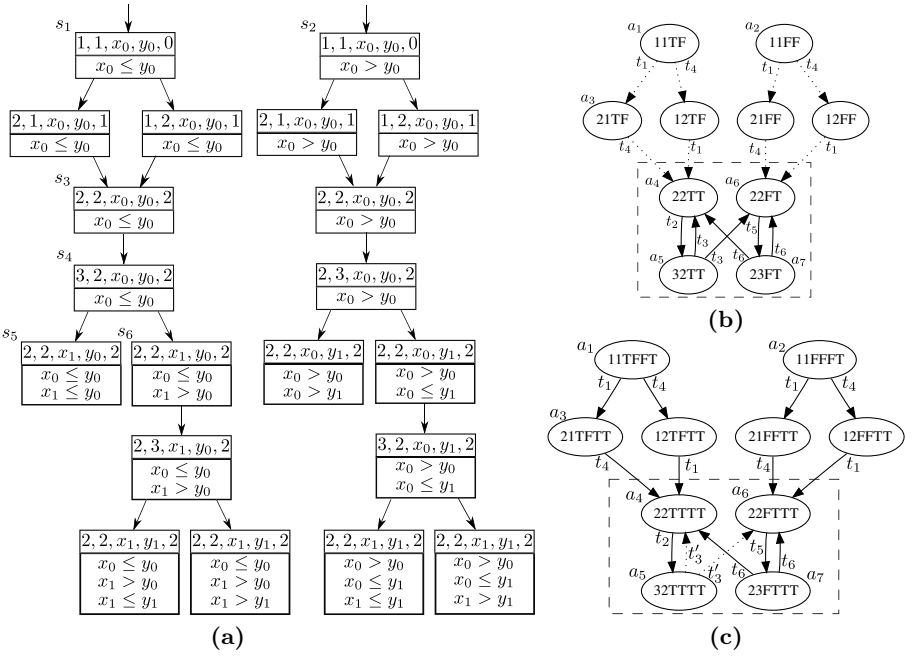


Fig. 3. (a) Symbolic execution of the program in Fig. 1; (b) its corresponding abstract transition system M_a; (c) a modified abstract transition system M'_a

The check succeeds in our example. This is easily verified by looking at Fig. 3(b), where all paths are trapped in the safe exact fragment consisting of the states a_4, a_5, a_6, and a_7. Thus, the program in Fig. 1 satisfies the mutual exclusion property.

Analysis: inductive-invariant. This check determines whether the state-space of M_a is an inductive invariant: i.e., it is closed under applying transitions of P. If so, the state-space of M_a over-approximates that of P, and thus P is safe. This check is complimentary to safe-fragment described above (see Section 4.2). If it fails, we move to the refinement phase.

Refinement. In this phase, we generate new predicates to refine inexact transitions of M_a. The refinement is based on computing preimage and is similar to the commonly used weakest precondition-based refinement. Although not needed in our running example, we illustrate refinement using the inexact transition $a_1 \xrightarrow{t_1} a_3$ of M_a in Fig. 3(b). First, we compute the preimage of a_3 w.r.t. transition t_1, resulting in $(pc_2 = 2 \wedge x \leq y \wedge b \neq 1)$. Second, we add only the predicate $b \neq 1$ to Φ_0 since program counter pc_2 is represented explicitly, and we already have $x \leq y$.

In the remainder of the paper, we formalize the above notions and evaluate the efficiency of our approach.

3 Preliminaries

This section outlines the definitions and notation used in this paper.

Program. We use a guarded command language to specify programs. A *program* P is a tuple (V, I, T), where V is a finite set of integer variables, $I(V)$ is an initial condition, and T is a finite set of transitions. Each transition $t \in T$ is of the form $g_t \longrightarrow e_t$, where g_t is a Boolean expression over the variables V, and e_t is a set of concurrent assignments. Each assignment is of the form $x := linExp$ or $x := nondet$, where x is a variable in V, $linExp$ is an expression from linear arithmetic over variables in V, and $nondet$ is a special expression used to denote non-deterministic input.

Transition System. A *transition system* over a finite set of atomic propositions AP and a set of transition labels T is a tuple (S, R, S_0, L), where S is a (possibly infinite) set of states, $R \subseteq S \times T \times S$ is the transition relation, $S_0 \subseteq S$ is the set of initial states, and $L : S \to 2^{AP}$ is a labelling function, mapping each state to the set of atomic propositions that hold in it. For clarity, we write $s \xrightarrow{t} s'$ to denote $R(s, t, s')$.

The *concrete semantics* of a program $P = (V, I, T)$ is a transition system $C(P) = (S, R, S_0, L)$ over some atomic propositions AP and the set of program transitions T, where $S = 2^{V \to \mathbb{Z}}$, $S_0 = \{s \in S \mid s \models I\}$, and $s \xrightarrow{t} s'$ for some $t \in T$ iff $s \models g_t$ and $s' \in e_t(s)$. By $s \models g_t$, we mean that the valuation of variables in s satisfies the Boolean expression g_t, and $e_t : (V \to \mathbb{Z}) \to 2^{(V \to \mathbb{Z})}$ is a function which computes all possible states resulting from applying the assignments to some state. Finally, $L(s) = \{\phi \in AP \mid s \models \phi\}$.

Preimage and Strongest Postcondition. Let ϕ be a formula over program variables. The *preimage* of ϕ w.r.t. a transition t, $pre(\phi, t) = \exists s' \cdot (s \xrightarrow{t} s' \wedge s' \models \phi)$, is a formula describing the set of all states which can reach a state satisfying ϕ via t. The *strongest postcondition* of ϕ w.r.t. a transition t, $sp(\phi, t) = \exists s' \cdot (s' \xrightarrow{t} s \wedge s' \models \phi)$, is a formula describing the set of all states that are reachable via t from a state satisfying ϕ.

Predicate Abstraction. Let $\Phi = \{\phi_1, \cdots, \phi_n\}$ be a set of predicates over program variables. The *predicate abstraction* α_Φ is a function from concrete states to Boolean formulae (abstract states) over predicates in Φ. Given a concrete state s, $\alpha_\Phi(s) = \bigwedge_{\phi \in \Phi_s} \phi \wedge \bigwedge_{\phi \in \overline{\Phi}_s} \neg\phi$, where $\Phi_s = \{\phi \in \Phi \mid s \models \phi\}$ and $\overline{\Phi}_s = \Phi \setminus \Phi_s$. A *concretization function* γ_Φ takes a Boolean formula over Φ and returns the set of concrete states satisfying the formula. Given a Boolean formula ψ over Φ, $\gamma_\Phi(\psi) = \{s \in S \mid s \models \psi\}$. For a set of states X, we write $\alpha_\Phi(X)$ to mean $\bigvee\{\alpha_\Phi(s) \mid s \in X\}$.

A transition $a_1 \xrightarrow{t} a_2$, where a_1 and a_2 are abstract states is a *must* transition iff $\forall s \in \gamma_\Phi(a_1) \cdot \exists s' \in \gamma_\Phi(a_2)$ s.t. $s \xrightarrow{t} s'$. A transition is a *may* transition iff $\exists s \in \gamma_\Phi(a_1) \cdot \exists s' \in \gamma_\Phi(a_2)$ s.t. $s \xrightarrow{t} s'$. In this paper, we call *must* transitions *exact*, and transitions that are *may* but not *must* – *inexact*. A transition $a_1 \xrightarrow{t} a_2$ is exact iff $a_1 \Rightarrow pre(a_2, t)$.

```
1: function Refine(P, ψ)
2:     Φ ← predicates from guards in P and ψ
3:     while true do
4:         inductive ← true
5:         (fin, inf, A0) ← symbolicExec(P, Φ)                    ▷ symbolic execution
6:         if a state in (fin, inf, A0) satisfies ¬ψ then return false
7:         if safeFragment(fin, inf) then return true              ▷ safe-fragment
8:         A ← all states in (inf, fin, A0)
9:         for all (a1, t, a2) ∈ (fin ∪ inf) do                    ▷ inductive-invariant
10:            if ¬(a1 ⇒ pre(a2, t)) then
11:                add predicates in pre(a2, t) to Φ                ▷ refinement
12:                if ¬(sp(a1, t) ⇒ ⋁ A) then inductive ← false
13:        if inductive then return true
```

Fig. 4. Refinement loop (main function)

4 Abstract Analysis of Symbolic Executions

In this section, we describe our algorithm in detail and discuss its properties.

4.1 Algorithm

Our abstraction-refinement based verification algorithm is implemented by the function Refine (Fig. 4) which does symbolic execution followed by safe-fragment and inductive-invariant checks and refinement (see Fig. 2). It uses two helper functions: symbolicExec (Fig. 5), to do symbolic execution with abstract matching and to compute the explored abstract transition system, and safeFragment (Fig. 6), to prove safety of the abstract transition system.

Refine initializes the set Φ with all the predicates in the program's guards and in the safety property ψ (line 2), and enters the execute-analyse-refine loop (lines 3–13). It uses symbolicExec (line 5) to compute the abstract transition system. It terminates with *false* if an error state is found (line 6); otherwise, it performs the safe-fragment check (line 7) followed, if needed, by the inductive-invariant check (lines 9–13). The Boolean variable *inductive* holds the result of inductive-invariant. If it is *false* after inductive-invariant (line 13), then Refine repeats symbolic execution with new predicates added to Φ; otherwise, it returns *true*.

symbolicExec performs depth-first symbolic execution with abstract matching. It uses the stack *symbStack* of *sets* of symbolic states to keep track of the current path. A *symbolic state* s over a set of variables V is a tuple (f, PC), where f is a function mapping each program variable to an integer or symbolic constant, and a *path condition* PC is a set of constraints over symbolic and integer constants. A concrete state c is represented by a symbolic state $s = (f, PC)$ iff $c \models \exists V_s \cdot \bigwedge_{y \in V} y = f(y) \wedge \bigwedge_{p \in PC} p$, where $V_s = \{z \mid \exists y \cdot f(y) = z \wedge z \text{ is a symbolic constant}\}$. For example, the state $(\{x \mapsto x_0, y \mapsto y_0\}, \{y_0 > 0, x_0 > y_0\})$ denotes the set of concrete states where y is strictly greater than 0 and x is strictly greater than y.

Let x be a variable in V and $s = (f, PC)$ a symbolic state. The symbolic execution is done by the function Exec (line 14 of Fig. 5) using the following rules: if e_t is $x := nondet$ then the result is the state $s' = (f[x \rightarrow z], PC)$,

```
1: function SYMBOLICEXEC(P, Φ)
2:     symbStack ← empty stack                ▷ Each item on the stack is a set of states
3:     push SPLITSTATE(s_0, Φ) on symbStack
4:     A_0 ← {α_Φ(s) | s ∈ SPLITSTATE(s_0, Φ)}
5:     (fin,inf,trans) ← (∅, ∅, ∅)
6:     while symbStack is not empty do
7:         S ← top of symbStack
8:         choose s ∈ S s.t. for some t, s ⊨ g_t
9:         if no such transition exists or transitions exhausted then
10:            fin ← fin ∪ ALLPATHS(S, trans)
11:            trans ← tail of trans
12:            pop symbStack
13:            continue
14:        result ← EXEC(s, e_t)
15:        S' ← SPLITSTATE(result, Φ)
16:        for all {s' ∈ S' | α_Φ(s') = α_Φ(s) or ∃t · (α_Φ(s'), t') ∈ trans} do
17:            fin ← fin ∪ STEM((α_Φ(s), t, α_Φ(s')), trans)
18:            inf ← inf ∪ LOOP((α_Φ(s), t, α_Φ(s')), trans)
19:            S' ← S' \ {s'}
20:        if S' ≠ ∅ then
21:            push S' on symbStack
22:            trans ← prepend (α_Φ(s), t) to trans
23:    return (fin, inf, A_0)
```

Fig. 5. Symbolic execution with abstract matching

```
1: function SAFEFRAGMENT(fin,inf)
2:     worklist ← inf
3:     while worklist ≠ ∅ do
4:         (a, t, b) ← remove element from worklist
5:         if ¬(a ⇒ pre(b, t)) then return false
6:         T ← {(a', t', b') ∈ fin | a' ∈ {a, b}}
7:         fin ← fin \ T
8:         worklist ← worklist ∪ T
9:     return true
```

```
function SPLITSTATE(s)
    S ← ∅, X ← α_Φ(s)
    for all minterms x ∈ X do
        S ← S ∪ (f, PC ∪ x[f(v_1)/v_1, ··· , f(v_n)/v_n])
    return S
```

Fig. 6. Algorithms for safe-fragment check and splitting symbolic states

where z is fresh symbolic constant (i.e., is not used in any symbolic state so far) and $f[x \to z](a)$ is z if $x = a$ and $f(a)$ otherwise; if e_t is $x := u$ for some expression u, then the result is the state $s' = (f[x \to z], PC')$, where z is fresh and $PC' = PC \cup (z = u[f(v_1)/v_1, \cdots, f(v_n)/v_n])$; if e_t is a concurrent assignment, the result is obtained by the obvious generalization of the base rules.

Recall that we require that a symbolic state satisfies or refutes each predicate in Φ. We enforce this using SPLITSTATE that takes a symbolic state as input and returns a set of states that satisfy our constraint. The naïve implementation is to recursively split a state s by taking a predicate from $p \in \Phi$ that is neither satisfied nor refuted by s and creating two new states by adding p and $\neg p$ to the path constraint of s, respectively. This is highly inefficient.

Instead, we reduce the problem to predicate abstraction as shown in Fig. 6. Basically, we compute a predicate abstraction of a symbolic state s and then split s using all the minterms in the result[1]. For example, consider the symbolic state $s = (\{x \mapsto x_0, y \mapsto y_0\}, \{y_0 = 0\})$ over $V = \{x, y\}$, and let $\Phi = \{x >$

[1] A minterm is a conjunction of literals containing all predicates in Φ.

$0, y > 0\}$. Predicate abstraction of s over Φ has two minterms: $\{x > 0 \wedge y \leq 0, x < 0 \wedge y \leq 0\}$. This leads to two new symbolic states $\{(f', PC'), (f'', PC'')\}$, where $f'' = f' = \{x \mapsto x_0, y \mapsto y_0\}$, $PC' = \{y_0 = 0, x_0 > 0, 0 \leq 0\}$, and $PC'' = \{y_0 = 0, x_0 \leq 0, 0 \leq 0\}$. Of course, tautologies like $0 \leq 0$ are discarded when the expression is simplified. This has the same worst-case complexity as the naïve approach, but allows us to use recent advances in predicate abstraction (such an AllSAT SMT solver).

In SYMBOLICEXEC, the variable *trans* keeps an abstraction of the path from the initial state to states at the top of *symbStack* as a list of tuples (a, t), where a is an abstract state and t a transition. Whenever SYMBOLICEXEC reaches a state whose abstraction has been seen before on the current path (i.e., either it is the same as a predecessor or it appears in *trans* – line 16), it stops current exploration and backtracks. At this point, *trans* is a lasso-shaped abstract path. Functions STEM and LOOP are used to extract the transitions that occur on the stem of the path, stored in set *fin* and the loop of the path, stored in *inf*. For example, consider the symbolic execution tree shown in Fig. 3(a). When SYMBOLICEXEC takes the transition $s_4 \xrightarrow{t_3} s_5$ (corresponding to the abstract transition $a_5 \xrightarrow{t_3} a_4$ in the abstract model in Fig. 3(b)) it discovers a loop. Then, $trans = [(a_4, t_2), (a_3, t_4), (a_1, t_1)]$, $\text{STEM}(trans, (a_5, t_3, a_4)) = \{(a_1, t_1, a_3), (a_3, t_4, a_4)\}$, and $\text{LOOP}(trans, (a_5, t_3, a_4)) = \{(a_4, t_2, a_5), (a_5, t_3, a_4)\}$. Note that transitions in the list *trans* are stored in reverse order from which they appear on the abstract path.

When SYMBOLICEXEC reaches a set of symbolic states where no transition can be taken (line 9), the transitions leading to that set of states are added to *fin* using the ALLPATHS function. For example, assume $trans = [(a_2, t_2), (a_1, t_1)]$ and $S = \{a_3, a_4\}$. Then, $\text{ALLPATHS}(S, trans) = \{(a_1, t_1, a_2), (a_2, t_2, a_3), (a_2, t_2, a_4)\}$.

SAFEFRAGMENT works by locating the fragment of the abstract model that is reached by all execution paths. This is done by finding all transitions in or reachable from *inf*. If all of these transitions are exact, then we conclude safety. Otherwise, we proceed to inductive-invariant.

In inductive-invariant (lines 9–13 of REFINE), for every state which is the source of an inexact transition t, we check if its strongest postcondition w.r.t. t is a subset of the set of abstract states explored (line 12). If so, the explored abstract states over-approximate the set of reachable concrete states, and thus we can conclude safety. Otherwise, we go back to the symbolic execution stage, but now with new predicates added from preimages of destination states of inexact transitions (line 11 of REFINE).

4.2 Soundness and Monotonicity

Our algorithm only reports real errors. This is ensured by restricting symbolic execution to explore only symbolic states with satisfiable path constraints. Theorem 1 states that the algorithm is also sound for safety properties[2]. Of course, since property checking is undecidable, the algorithm is incomplete.

[2] Proofs of all theorems can be found in [1].

In the rest of this section, we represent the abstract state-space explored by SYMBOLICEXEC by a transition system $M_a = (S^a, R^a, S_0^a, L^a)$, where $AP = \Phi$, S^a is the set of all states appearing in (fin, inf, A_0), R^a is the set of all transitions appearing in $fin \cup inf$, $S_0^a = A_0$, and for $x \in S^a$, $L^a(x) = \{\phi \in \Phi \mid x \models \phi\}$.

Theorem 1 (Soundness). *Let ψ be a safety property, P be a program satisfying ψ, and $M_c = (S, R, S_0, L)$ be a transition system of P. W.l.o.g., assume that every state in S is reachable from S_0. Let $M_a = (S^a, R^a, S_0^a, L^a)$ be the abstract transition system constructed by* SYMBOLICEXEC *in the last iteration of* REFINE. *Then, (i) if* REFINE *terminates after* safe-fragment *(line 7), then M_a simulates M_c; (ii) if* REFINE *terminates after* inductive-invariant *(line 13), then S^a over-approximates S (i.e., $\forall s \in S \cdot \alpha_\Phi(s) \in S^a$).*

As in SYNERGY (see Sec. 4 in [15]), when our algorithm terminates with safe-fragment, the current abstraction simulates, but is not necessarily bisimular to, the concrete program. Moreover, if it terminates with inductive-invariant then the abstraction may not even simulate the concrete program.

In contrast with other under-approximating approaches, e.g., [23,4], our algorithm explores more states in each successive iteration than in a previous one. That is, the exploration is monotonically increasing. This ensures steady progress towards an error state (if one exists). Intuitively, we get this by keeping an abstract visited table per each path, as opposed to a unique global table as in [23].

Theorem 2 (Monotonicity). *Let Φ and Φ' be two sets of predicates s.t. $\Phi \subseteq \Phi'$. Let P be a program, and C and C' be the concrete states of P explored by* SYMBOLICEXEC *under Φ and Φ', respectively. Then, $C \subseteq C'$.*

In contrast, our approach is not monotonic for proving safety: adding new predicates may cause an exact transition used by safe-region check to become inexact [12,23,4]. In the future, we hope to solve this problem by using an abstract domain of tri-vectors.

As discussed in Section 2, the two checks, safe-fragment and inductive-invariant, are incomparable. We prove this below.

Theorem 3. *There is an abstract model M_a constructed by* SYMBOLICEXEC *that passes exactly one of* safe-fragment *and* inductive-invariant *checks.*

Proof. First, we give an example where safe-fragment holds but inductive-invariant fails. Consider M_a in Fig. 3(b). Recall that it passes safe-fragment check. It fails inductive-invariant since it is not closed under strongest postcondition: $sp(a_1, t_1) = (pc_1 = 2 \wedge pc_2 = 1 \wedge x \leq y \wedge b \neq 2) \vee (pc_1 = 2 \wedge pc_2 = 1 \wedge x \leq y \wedge b = 2)$; the second disjunct is not covered by an explored abstract state.

Second, we give an example where inductive-invariant holds but safe-fragment fails. Consider M'_a shown in Fig. 3(c). It is obtained from symbolically executing a program obtained by replacing transition t_3 by $t_3 : pc_1 = 3 \longrightarrow pc_1 := 2,\ x := x + 1$ in the protocol in Fig. 1, and assuming that Φ includes predicates $b = 0$, $b = 1$, and predicates from the guards. All transitions of M'_a, with the exceptions

		Iter		Prvr. Qurs.		Preds.		Time(s)		Con/Abs States		Check
Program	ψ	ASE	UR	ASE	UR	ASE	UR	ASE	UR	ASE	UR	ASE
bakery$_2$	t	3	4	141	367	8	10	0.347	0.452	52/33	49/36	II
RAX	t	1	-	6	-	2	-	0.261	-	81/44	-	SF
elev$_4$	t	1	4	418	5789	13	19	1.013	8.146	468/378	468/456	SF
elev$_5$	t	1	5	1169	26252	15	23	3.459	44	1256/910	1253/1204	SF
elev$_6$	t	1	6	3156	105830	17	27	12.275	220.633	3248/2126	3224/3060	SF
elev$_7$	t	1	-	7116	-	19	-	40.867	-	8160/4862	-	SF
elev$_8$	t	1	-	15036	-	21	-	185.717	-	15200/9422	-	SF
ticket$_2$	t	4	4	135	120	8	8	0.609	0.404	22/9	12 / 9	SF
ticket$_3$	t	5	5	672	661	14	14	1.413	0.923	182/31	41/31	SF
ticket$_4$	t	6	6	4088	4061	23	23	33.51	5.143	5011/129	170/129	SF
mesi	t	16	16	6893	12172	47	47	36.61	49.627	18/18	18/18	SF
berkley	t	11	11	3113	4623	38	38	15.729	17.605	13/12	13/12	SF
b_bakery$_2$-e	f	1	2	0	74	2	5	0.178	1.188	80/80	193/193	-
ticket$_2$-e	f	1	2	0	11	2	5	0.073	0.155	12 / 9	26 / 17	-
ticket$_3$-5	t	1	3	0	145	3	14	0.058	0.341	14/12	93/81	-
ticket$_3$-10	t	3	8	152	1218	14	21	0.525	2.229	30/27	302/240	-
ticket$_3$-15	t	8	13	1225	2090	21	26	3.107	5.15	47/44	507/395	-
ticket$_3$-20	t	13	18	2500	3918	26	31	6.869	9.501	62/59	712/550	-
ticket$_3$-25	t	18	23	3925	5493	31	36	13.038	15.821	77/74	917/705	-
ticket$_3$-30	t	23	28	5500	7219	36	41	20.762	34.701	92/89	1112/860	-
ticket$_3$-35	t	28	33	7225	9093	41	46	46.379	51.579	107/104	1327/1015	-
ticket$_3$-40	t	33	38	9100	11118	46	51	71.462	82.974	122/119	1532/1170	-
RAX-5	t	5	5	46	123	12	20	0.373	0.363	50/49	170/170	-
RAX-10	t	10	10	146	483	17	35	0.988	1.528	90/89	350/350	-
RAX-15	t	15	15	296	1068	22	50	2.031	4.341	130/129	530/530	-
RAX-20	t	20	20	496	1878	27	65	3.675	9.934	170/169	710/710	-
RAX-25	t	25	25	746	2913	32	80	6.442	19.578	210/209	890/890	-
RAX-30	t	30	30	1046	4173	37	95	9.94	35.03	250/249	1070/1070	-
RAX-35	t	35	35	1396	5658	42	110	15.155	57.315	290/289	1250/1250	-
RAX-40	t	40	40	1796	7368	47	125	22.104	89.332	320/319	1430/1430	-
RAX-45	t	45	45	2246	9303	52	140	30.821	133.063	370/369	1610/1610	-

Fig. 7. Experimental results: ASE vs. UR [23]

of the two transitions from a_5, are exact. safe-fragment fails on M'_a. inductive-invariant does not: the only interesting case is that $sp(a_5, t'_3) = (pc_1 = 2 \wedge pc_2 = 2 \wedge b = 2)$ is covered by explored abstract states. □

We have shown that our algorithm is sound and explores the concrete state-space monotonically. We have also shown that the two safety checks, safe-fragment and inductive-invariant, are incomparable. Hence, both are useful.

5 Implementation and Experimental Results

We have implemented our algorithm in OCaml on top of the implementation of Pasareanu et al. [23]. We used GiNaC [5] for symbolic execution, MathSAT4 [6] for computing predicate abstraction, Simplify [10] for checking exactness of transitions and computing inductive invariants, and Bradley's implementation of Cooper's method for quantifier elimination[3]. In all of our experiments, we added predicates only from those inexact transitions that are in the set *inf* (returned by symbolicExec) or reachable from it.

[3] Available at http://theory.stanford.edu/~arbrad/sware.html

Program	ψ	Iter.	Prvr. Qurs.	Preds.	Symb/Abs States	Time(s)	Check
ticket$_2$	t	4	523	12	5516/62	281.134	II
peterson$_2$	t	1	24	4	700/38	424	II
bakery$_2$	t	3	301	11	807/50	73.965	II
mesi	t	2	260	13	112/14	3.56	II
synapse	t	2	62	7	34/9	0.88	II
ticket$_2$-e	f	1	0	2	12/10	0.104	-
ticket$_3$-e	f	1	0	3	10/10	0.112	-

Fig. 8. Experimental results: programs with unspecified initial states and non-deterministic input

In Fig. 7, we compare effectiveness of our *abstract analysis of symbolic executions* approach (referred to as ASE) with that of the *under-approximation refinement* algorithm of [23] (referred to as UR). We indicate whether the safety property of interest (ψ) is true (t) or false (f) and report the number of iterations (Iter.), the number of theorem prover queries (Prvr. Qurs.), the total number of predicates used (Preds.), the total amount of time needed, the number of concrete and abstract states explored in the final iteration, and the check with which ASE concluded safety (II for inductive-invariant, SF for safe-fragment, and "–" when a counterexample is returned). In cases where the experiment did not finish after 15 minutes, the table entries are "–".

Since UR can only handle a single concrete initial state and no non-deterministic input, these are the characteristics of all programs in Fig. 7. We began by checking the mutual exclusion property of the `bakery` protocol with two processors, where our performance is a bit better than UR. On the other hand, ASE can prove that the Remote Agent Experiment (RAX), as presented in [23], is deadlock-free in a single iteration, while UR refines indefinitely. We then verified the elevator program, `elev`$_i$, increasing the number of floors i, against the property that the elevator cannot be on two separate floors at the same time. We checked mutual exclusion of the `ticket`$_i$ protocol, increasing the number of processes i, as well as correctness cache coherence protocols `mesi` and `berkley` (these, along with their correctness properties, are taken from [9], restricting the number of initial states to one). Our results show that ASE generally outperforms UR in terms of the number of iterations and time it takes to prove safety. In the case of `ticket`$_i$ where ASE requires the same number of iterations and predicates, ASE takes more time as it explores more concrete states per iteration.

To illustrate the power of our approach at finding errors, we analysed defective versions, i.e., not satisfying mutual exclusion, of the bounded bakery (`b_bakery`$_2$`-e`) and ticket (`ticket`$_2$`-e`) protocols. We also checked whether a given ticket number X in the `ticket`$_i$ protocol (`ticket`$_i$`-`X) and a given counter value X in the `RAX` example (`RAX-`X) are reachable. ASE terminates in fewer iterations than UR in the former case and in the same number of iterations but significantly fewer predicates in the latter.

In Fig. 8, we report on the results of ASE for checking properties of programs with unspecified initial states and/or non-deterministic input. Specifically, we verified mutual exclusion of `ticket`, where the initial ticket number is set non-deterministically, and `bakery` and `peterson` protocols, where each process stays

in the critical section for a non-deterministic amount of time. We also verified correctness of cache coherence protocols, `mesi` and `synapse`, with undefined initial states.

In summary, ASE can analyse a wide range of programs that manipulate arbitrary integers and use non-deterministic input. And it can do so in less time, and considerably fewer iterations or with significantly fewer predicates than UR.

6 Related Work

The work by Pasareanu et al. [23] is the closest to ours. However, there are several key differences. First, our approach explores the state-space monotonically. Second, we use symbolic execution to deal with programs with arbitrary initial states and non-deterministic input. Third, we use over-approximation much more aggressively leading to a much faster convergence with fewer predicates. Comparison with other work is given below.

Over-approximation based techniques (e.g., [3,16,7]) build an abstraction that has more behaviours than the concrete system and prune infeasible computations via refinement. In contrast, our refinement is based on extending the frontier of feasible program behaviours. Most of such techniques, with the exception of [7], deal with sequential programs only.

Under-approximation based techniques [23,4,20,24] build an abstraction that has fewer program behaviours than the concrete system. Our approach includes both reachable *must* and *may* transitions making the abstract models more precise than those that have just *must* transitions (e.g., [24]) and *must* and *reverse must* transitions (e.g., [4][4]). The algorithm in [20] builds a finite bisimiluation quotient of the program under analysis, but unlike the global refinement employed by us and [23], uses a local refinement instead. We leave a comparison of the efficiency of local and global refinements for future work.

Most recent automated software verification techniques that combine dynamic analysis for detecting bugs and static analysis for proving correctness (e.g., [26,15,13,19]) concentrate on analysis of sequential programs, and unlike our approach which bounds program executions, assume terminating program executions. For example, [26] uses tests cases to explore an under-approximating abstract state-space with the hope of exploring all reachable abstract states but has no notion of refinement and thus the analysis may return false positives. Like our work, [2] uses abstraction to bound symbolic execution of programs. While this approach can handle programs with recursive data structures and arrays, its goal is debugging rather than verification, and it does not involve refinement.

[19] improves error detection capabilities of the CEGAR framework [8] by using program execution to drive abstraction-refinement. However, it does so by refining an over-approximation and is restricted to sequential programs.

Directed automated random testing (DART) [14] and its successors, [25,11], run the program with random input, using path constraints to discover input that would exercise alternative program paths. The SYNERGY algorithm [15]

[4] see [1] for a detailed comparison with [4].

combines DART-like testing with over-approximating abstractions, using results of tests to refine the abstract model and using the abstract model to drive test case generation. The end result is either a test case that reaches an error state, or an abstract model that simulates the program. Whereas DART-like approaches attempt to cover all program paths, our approach and [23,26] attempt to cover all reachable abstract states. [13] presents a compositional algorithm that combines DART and over-approximating techniques. DART-like testing is used to create under-approximating (*must*) summaries of functions, and techniques based on [3] are used to create over-approximating (*may*) summaries. The authors show that alternating *must* and *may* summaries yields better results than *must* only or *may* only summaries. However, these techniques are restricted to sequential programs.

7 Conclusion and Future Work

We presented a novel verification algorithm that combines symbolic execution and predicate abstraction in an abstraction-refinement cycle. Our approach applies to concurrent programs with infinite data domain and non-deterministic input. Given a program and a safety property, our algorithm executes the program symbolically, while building an under-approximating abstract model. If an error is reached by symbolic execution, we terminate and report it. Otherwise, we check whether the state-space of the abstract model over-approximates all concretely reachable states. If the analysis fails, we refine with new predicates and repeat the process. Not only do we handle a much wider range of programs than related approaches, we also improve on the number of iterations and the number of predicates used, whether the property of interest is true or false.

Our current implementation is a proof of concept – more work is needed to turn it into robust verification tool that is applicable to a real programming language (such as C) with complex features (e.g., structured and recursive data types, pointers, recursion, etc.). It is also interesting to see whether the approach extends to termination (and non-termination) properties. A promising direction is to use the under-approximation to derive either a ranking function or a counterexample to termination. We leave exploring these for future work.

Acknowledgements. We would like to thank Corina Pasareanu and Radek Pelanek for giving us access to their code, the anonymous referees for their helpful comments, and the formal methods group at the University of Toronto for the fuitful discussions.

References

1. Albarghouthi, A.: Abstract Analysis via Symbolic Executions. Master's thesis, Univ. of Toronto, Dept. of Comp. Sci. (February 2010)
2. Anand, S., Pasareanu, C.S., Visser, W.: Symbolic Execution with Abstraction. STTT 11(1), 53–67 (2009)

3. Ball, T., Rajamani, S.: The SLAM Toolkit. In: Berry, G., Comon, H., Finkel, A. (eds.) CAV 2001. LNCS, vol. 2102, pp. 260–264. Springer, Heidelberg (2001)
4. Ball, T., Kupferman, O., Yorsh, G.: Abstraction for Falsification. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 67–81. Springer, Heidelberg (2005)
5. Bauer, C., Frink, A., Kreckel, R.: Introduction to the GiNaC Framework for Symbolic Computation with the C++ Programming Language. J. Symbolic Computation 33, 1–12 (2002)
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. Chaki, S., Clarke, E., Groce, A., Jha, S., Veith, H.: Modular Verification of Software Components in C. IEEE Tran. on Soft. Eng. 30(6), 388–402 (2004)
8. 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)
9. Delzanno, G.: Automatic Verification of Parameterized Cache Coherence Protocols. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 53–68. Springer, Heidelberg (2000)
10. Detlefs, D., Nelson, G., Saxe, J.: Simplify: a Theorem Prover for Program Checking. J. of the ACM 52(3), 365–473 (2005)
11. Godefroid, P.: Compositional Dynamic Test Generation. In: Proc. of POPL'07, pp. 47–54 (2007)
12. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based Model Checking using Modal Transition Systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 426. Springer, Heidelberg (2001)
13. Godefroid, P., Nori, A., Rajamani, S., Tetali, S.: Compositional May-Must Program Analysis: Unleashing the Power of Alternation. In: Proc. of POPL'10 (2010)
14. Godefroid, P., Klarlund, N., Sen, K.: DART: Directed Automated Random Testing. In: Proc. of PLDI'05. pp. 213–223 (2005)
15. Gulavani, B., Henzinger, T., Kannan, Y., Nori, A., Rajamani, S.: SYNERGY: a New Algorithm for Property Checking. In: Robshaw, M.J.B. (ed.) FSE 2006. LNCS, vol. 4047, pp. 117–127. Springer, Heidelberg (2006)
16. Henzinger, T., Jhala, R., Majumdar, R., Sutre, G.: Lazy Abstraction. In: Proc. of POPL'02. pp. 58–70 (January 2002)
17. Holzmann, G., Joshi, R.: Model-Driven Software Verification. In: Graf, S., Mounier, L. (eds.) SPIN 2004. LNCS, vol. 2989, pp. 76–91. Springer, Heidelberg (2004)
18. Holzmann, G.: The Model Checker SPIN. IEEE Tran. on Soft. Eng. 23(5) (1997)
19. Kroening, D., Groce, A., Clarke, E.: Counterexample Guided Abstraction Refinement via Program Execution. In: Davies, J., Schulte, W., Barnett, M. (eds.) ICFEM 2004. LNCS, vol. 3308, pp. 224–238. Springer, Heidelberg (2004)
20. Lee, D., Yannakakis, M.: Online Minimization of Transition Systems. In: Proc. of STOC'92. pp. 264–274 (1992)
21. Musuvathi, M., Qadeer, S.: CHESS: Systematic Stress Testing of Concurrent Software. In: Puebla, G. (ed.) LOPSTR 2006. LNCS, vol. 4407, pp. 15–16. Springer, Heidelberg (2007)
22. Nori, A., Rajamani, S., Tetali, S., Thakur, A.: The Yogi Project: Software Property Checking via Static Analysis and Testing. In: TACAS 2009. LNCS, vol. 5505, pp. 178–181. Springer, Heidelberg (2009)

23. Pasareanu, C., Pelanek, R., Visser, W.: Concrete Model Checking with Abstract Matching and Refinement. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 52–66. Springer, Heidelberg (2005)
24. Pasareanu, C., Dwyer, M., Visser, W.: Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 284–298. Springer, Heidelberg (2001)
25. Sen, K., Marinov, D., Agha, G.: CUTE: A Concolic Unit Testing Engine for C. In: Proc. of ESEC/FSE'05, pp. 263–272 (2005)
26. Yorsh, G., Ball, T., Sagiv, M.: Testing, Abstraction, Theorem Proving: Better Together! In: Proc. of ISSTA'06. pp. 145–156 (2006)

Automated Assume-Guarantee Reasoning through Implicit Learning*

Yu-Fang Chen[1], Edmund M. Clarke[2], Azadeh Farzan[3], Ming-Hsien Tsai[4], Yih-Kuen Tsay[4], and Bow-Yaw Wang[1,5]

[1] Academia Sinica, Taiwan
[2] Carnegie Mellon University, USA
[3] University of Toronto, Canada
[4] National Taiwan University, Taiwan
[5] INRIA, France and Tsinghua University, China

Abstract. We propose a purely implicit solution to the contextual assumption generation problem in assume-guarantee reasoning. Instead of improving the L^* algorithm — a learning algorithm for *finite automata*, our algorithm computes implicit representations of contextual assumptions by the CDNF algorithm — a learning algorithm for *Boolean functions*. We report three parametrized test cases where our solution outperforms the monolithic interpolation-based Model Checking algorithm.

1 Introduction

Assume-guarantee reasoning is a divide-and-conquer technique to alleviate the state explosion problem in formal verification. Let M be a transition system and π a predicate on states of M. We write $M \models \pi$ to denote that all reachable states of M satisfy the state predicate π. The composition of transition systems M and M' is denoted by $M \| M'$. Moreover, $M \preceq M'$ means that M is simulated by M'. Consider the following assume-guarantee reasoning rule:

$$\frac{M_0 \| A \models \pi \qquad M_1 \preceq A}{M_0 \| M_1 \models \pi}$$

In order to prove that the composition of M_0 and M_1 satisfies π, it suffices to find a transition system A such that the composition of M_0 and A satisfies the state

* This research was sponsored by the GSRC under contract no. 1041377 (Princeton University), National Science Foundation under contracts no. CCF0429120, no. CNS0926181, no. CCF0541245, and no. CNS0931985, Semiconductor Research Corporation under contract no. 2005TJ1366, General Motors under contract no. GM-CMUCRLNV301, Air Force (Vanderbilt University) under contract no. 18727S3, the Office of Naval Research under award no. N000141010188, the National Science Council of Taiwan projects no. NSC97-2221-E-001-003-MY3, no. NSC97-2221-E-001-006-MY3, no. NSC97-2221-E-002-074-MY3, no. NSC99-2218-E-001-002-MY3, and iCAST under contract no. 1010717, Natural Sciences and Engineering Research Council of Canada NSERC Discovery Award, the FORMES Project within LIAMA Consortium, and the French ANR project SIVES ANR-08-BLAN-0326-01.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 511–526, 2010.

predicate π, and that M_1 is simulated by A. Informally, the transition system A captures necessary assumptions about the context of M_0 to guarantee π. We thus call A a contextual assumption. The contextual assumption generation problem is to compute a contextual assumption in an assume-guarantee reasoning rule.

We address the contextual assumption generation problem in this paper. In [11], the problem is formulated as an automata learning problem. The authors apply the L^* algorithm [1] to generate a deterministic finite automaton as the contextual assumption. In contrast to previous works [14,13,7,5,23,20,11], our solution does not rely on the L^* algorithm. Instead, we use the CDNF algorithm [4] to generate Boolean functions that implicitly represent contextual assumptions in assume-guarantee reasoning. One can think of the relation between our approach and L^*-based techniques as very similar to the relation between implicit and explicit Model Checking. Succinct implicit representations give our algorithm advantages in generating contextual assumptions of a moderate size. They hence make our solution more scalable and applicable.

Our new technique directly computes implicit representations of contextual assumptions by applying the CDNF algorithm [4]. The CDNF algorithm is an exact learning algorithm for arbitrary Boolean functions. It assumes an active learning model similar to that in the L^* algorithm [1]. In its learning model, a membership query asks a teacher if a valuation satisfies the target Boolean function. An equivalence query asks if a conjecture is equivalent to the target Boolean function. If not, the teacher should give a counterexample so that the learning algorithm can refine the conjecture. The CDNF algorithm is a feasible learning algorithm. It infers any target Boolean function with a polynomial number of queries in the size of the target function and the number of variables [4].

In [11], all components and the contextual assumption were modeled as finite automata. The contextual assumption generation problem was solved by learning a deterministic finite automaton as the contextual assumption. In contrast, we view the problem as a Boolean function learning problem. In our setting, transition systems and hence contextual assumptions are implicitly represented by Boolean functions. The simulation relation $M_1 \preceq A$ in the assume-guarantee reasoning rule gives a simple characterization of the Boolean functions representing the transition system M_1 and a contextual assumption A. We thus exploit the information to resolve membership queries. Moreover, the premise $M_0 \| A \models \pi$ in the assume-guarantee reasoning rule further characterizes the Boolean functions representing the transition system M_0 and the contextual assumption A. This allows us to resolve equivalence queries in our algorithm.

It is important to note that our algorithm is not an optimization of the explicit L^* algorithm in any way. Instead, our algorithm simply generates contextual assumptions implicitly by employing an exact learning algorithm for Boolean functions. The most significant advantage of our solution is its scalability. This can be observed in two aspects. Recall that the L^* algorithm requires a polynomial number of queries in the *number of states* of the target finite automaton [1,21]. The CDNF algorithm, on the other hand, requires a polynomial number of queries in the *number of Boolean variables* of the target Boolean

function [4]. Since implicit representations obtained in our algorithm can be exponentially more succinct than explicit ones obtained in automata-theoretic algorithms, our solution can be exponentially better than explicit algorithms.

Comparing the qualities of generated contextual assumptions, our solution is also favorable. Most existing automata-theoretic algorithms are based on variants of the L^* algorithm [1,21], they inherently generate deterministic finite automata as contextual assumptions. In contrast, contextual assumptions generated by our algorithm are represented by general Boolean functions. In general, they are nondeterministic finite automata in an economic representation. Since nondeterministic finite automata can be exponentially more succinct than deterministic ones, our algorithm can generate contextual assumptions with exponentially less states than those generated by L^*-based algorithms. Even though implicit representations have been used in optimizing the L^* algorithm [23,13,20], our new implicit solution can still outperform these optimizations.

In [17], the CDNF algorithm is used to generate propositional loop invariants in sequential programs. The idea of using the L^* algorithm to learn contextual assumptions for assume-guarantee reasoning was first proposed in [11]. Following this work, there have been results for other assume-guarantee rules [2,20], symbolic implementations [20], various optimization techniques [6,23,15,7], performance evaluation [10], and extension to support liveness properties [12]. The common theme of these works is that they are all based on the L^* learning algorithm and hence always generate deterministic finite automata as contextual assumptions. To the best of our knowledge, the only exception is [3], which is essentially a modified version of the counterexample guided abstraction refinement technique [9]. Our solution is orthogonal to abstraction refinement; it can apply abstraction refinement techniques implemented in Model Checkers.

The paper is organized as follows. Section 2 gives the background of our presentation. We review the exact learning algorithm CDNF for Boolean functions in Section 3. It is followed by our solution to the contextual assumption generation problem (Section 4). Section 5 gives our preliminary experimental results. Finally, we conclude in Section 6.

2 Preliminaries

$\mathbb{B} = \{\mathsf{F}, \mathsf{T}\}$ is the Boolean domain. Let $\mathbf{x}$ be a set of Boolean variables and $|\mathbf{x}|$ the size of $\mathbf{x}$. A *Boolean function* $\theta(\mathbf{x})$ *over* $\mathbf{x}$ is a function from $\mathbb{B}^{|\mathbf{x}|}$ to $\mathbb{B}$. We also define $\mathbf{x}'$ to be the set of Boolean variables $\{x' : x \in \mathbf{x}\}$.

A *valuation* $\nu : \mathbf{x} \to \mathbb{B}$ *over* $\mathbf{x}$ is a function from Boolean variables to truth values. Let $\phi(\mathbf{x})$ be a Boolean function over $\mathbf{x}$ and ν a valuation over $\mathbf{x}$. If $\mathbf{y} \subseteq \mathbf{x}$ is a set of Boolean variables, $\nu{\downarrow_{\mathbf{y}}}$ is the *restriction* of ν on $\mathbf{y}$. That is, $\nu{\downarrow_{\mathbf{y}}}: \mathbf{y} \to \mathbb{B}$ and $\nu{\downarrow_{\mathbf{y}}}\ (y) = \nu(y)$ for all $y \in \mathbf{y}$. We write $\phi[\nu]$ for the result of evaluating ϕ by replacing each $x \in \mathbf{x}$ with $\nu(x)$. Moreover, let $\psi(\mathbf{x}, \mathbf{x}')$ be a Boolean function over $\mathbf{x}$ and $\mathbf{x}'$. If ν and ν' are valuations over $\mathbf{x}$, we write $\psi[\nu, \nu']$ for the result of evaluating ψ by replacing each $x \in \mathbf{x}$ with $\nu(x)$ and each $x' \in \mathbf{x}'$ with $\nu'(x)$. For example, assume $\nu(x) = \mathsf{F}$ and $\nu'(x) = \mathsf{T}$. If $\phi(x) = \neg x$, $\phi[\nu] = \mathsf{T}$ and $\phi[\nu'] = \mathsf{F}$. If $\psi(x, x') = \neg x \wedge x'$, $\psi[\nu, \nu'] = \mathsf{T}$ and $\psi[\nu', \nu] = \mathsf{F}$.

A *transition system* $M = (\mathbf{x}, \iota(\mathbf{x}), \tau(\mathbf{x}, \mathbf{x}'))$ consists of its *state variables* $\mathbf{x}$, its *initial predicate* $\iota(\mathbf{x})$, and its *transition relation* $\tau(\mathbf{x}, \mathbf{x}')$. A *trace of* M $\alpha = \nu^0\nu^1\cdots\nu^t$ is a finite sequence of valuations where ν^i is a valuation over $\mathbf{x}$, such that $\iota[\nu^0] = \mathsf{T}$ and $\tau[\nu^i, \nu^{i+1}] = \mathsf{T}$ for $0 \leq i < t$. Define $\mathit{Trace}(M) = \{\alpha : \alpha \text{ is a trace of } M\}$. If $\alpha = \nu^0\nu^1\cdots\nu^t$ is a finite sequence of valuations over $\mathbf{x}$ and $\mathbf{y} \subseteq \mathbf{x}$, $\alpha{\downarrow_{\mathbf{y}}} = \nu^0{\downarrow_{\mathbf{y}}}\ \nu^1{\downarrow_{\mathbf{y}}}\cdots\nu^t{\downarrow_{\mathbf{y}}}$ is the *restriction* of α on $\mathbf{y}$.

Let $M = (\mathbf{x}, \iota_M(\mathbf{x}), \tau_M(\mathbf{x}, \mathbf{x}'))$ be a transition system. A *state predicate* $\pi(\mathbf{x})$ is a Boolean function over $\mathbf{x}$. We say M *satisfies* π (denoted by $M \models \pi$) if for any $\alpha = \nu^0\nu^1\cdots\nu^t \in \mathit{Trace}(M)$, we have $\pi[\nu^i] = \mathsf{T}$ for $0 \leq i \leq t$. Given a transition system M and a state predicate π, the *invariant checking* problem is to decide whether M satisfies π. *Model Checking* is an automatic technique to solve the invariant checking problem. When deciding whether $M \models \pi$, a Model Checking algorithm returns a witness if M does not satisfy π. A *witness* to $M \not\models \pi$ is a trace $\nu^0\nu^1\cdots\nu^t$ of M such that $\pi(\nu^i) = \mathsf{T}$ for $0 \leq i < t$ but $\pi(\nu^t) = \mathsf{F}$.

Let $N = (\mathbf{x}, \iota_N(\mathbf{x}), \tau_N(\mathbf{x}, \mathbf{x}'))$ be a transition system. We say M *is simulated by* N or N *simulates* M (denoted by $M \preceq N$) if $\forall\mathbf{x}.\iota_M(\mathbf{x}) \implies \iota_N(\mathbf{x})$ and $\forall\mathbf{x}\mathbf{x}'.\tau_M(\mathbf{x}, \mathbf{x}') \implies \tau_N(\mathbf{x}, \mathbf{x}')$ hold. In words, M is simulated by N if the initial condition of M is more restrictive than that of N and every transition allowed in M is also allowed in N. Clearly, if $M \preceq N$, then $\mathit{Trace}(M) \subseteq \mathit{Trace}(N)$.

Let $\mathbf{x}_i$ be sets of Boolean variables for $i = 0, 1$ ($\mathbf{x}_i$'s are not necessarily disjoint). Consider $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}_i'))$ for $i = 0, 1$. The *composition* of M_0 and M_1 is the transition system $M_0\|M_1 = (\mathbf{x}_0 \cup \mathbf{x}_1, \iota_0(\mathbf{x}_0) \wedge \iota_1(\mathbf{x}_1), \tau_0(\mathbf{x}_0, \mathbf{x}_0') \wedge \tau_1(\mathbf{x}_1, \mathbf{x}_1'))$. Note that for any finite sequence of valuations α over $\mathbf{x}_0 \cup \mathbf{x}_1$, $\alpha \in \mathit{Trace}(M_0\|M_1)$ if and only if $\alpha{\downarrow_{\mathbf{x}_0}} \in \mathit{Trace}(M_0)$ and $\alpha{\downarrow_{\mathbf{x}_1}} \in \mathit{Trace}(M_1)$.

An *assume-guarantee reasoning rule* is of the form $\dfrac{\Theta_0\cdots\Theta_m}{\Delta}$ where $\Theta_0, \ldots, \Theta_m$ are its *premises* and Δ its *conclusion*. An assume-guarantee reasoning rule is *sound* if its conclusion holds when its premises are fulfilled. A rule is *invertible* if its premises can be fulfilled when its conclusion holds. We use the following assume-guarantee reasoning rule throughout the paper:

Lemma 1. *Let* $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}_i'))$ *be transition systems for* $i = 0, 1$, *and* π *a state predicate over* $\mathbf{x}_0 \cup \mathbf{x}_1$. *The following rule is sound and invertible:*

$$\frac{M_0\|A \models \pi \qquad M_1 \preceq A}{M_0\|M_1 \models \pi}$$

where $A = (\mathbf{x}_1, \iota_A(\mathbf{x}_1), \tau_A(\mathbf{x}_1, \mathbf{x}_1'))$ *is a transition system.*

Let $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}_i'))$ be transition systems for $i = 0, 1$ and π a state predicate over $\mathbf{x}_0 \cup \mathbf{x}_1$, a transition system $A = (\mathbf{x}_1, \iota_A(\mathbf{x}_1), \tau_A(\mathbf{x}_1, \mathbf{x}_1'))$ such that $M_0\|A \models \pi$ and $M_1 \preceq A$ is called a *contextual assumption* of M_0.

3 The CDNF Algorithm

For a fixed set of Boolean variables $\mathbf{x}$ and a Boolean function $\lambda(\mathbf{x})$ over $\mathbf{x}$, an exact learning algorithm for Boolean functions computes a representation of $\lambda(\mathbf{x})$

in a finite number of steps. The CDNF algorithm is an exact learning algorithm for Boolean functions [4]. Like the L^* algorithm [1], the CDNF algorithm uses an *active learning* model. In the model, it is assumed that *a teacher*, who knows the *target* Boolean formula $\lambda(\mathbf{x})$, provides the learning algorithm with answers to the following types of queries:

- *Membership query* $MEM(\nu)$ *for the target* $\lambda(\mathbf{x})$*,* where ν is a valuation over $\mathbf{x}$. If $\lambda[\nu] = \mathsf{T}$, the teacher answers *YES*; and *NO*, otherwise.
- *Equivalence query* $EQ(\theta)$ *for the target* $\lambda(\mathbf{x})$*,* where $\theta(\mathbf{x})$ is a Boolean function over $\mathbf{x}$. If the *conjecture* $\theta(\mathbf{x})$ is equivalent to the target Boolean function $\lambda(\mathbf{x})$, the teacher answers *YES*. Otherwise, the teacher provides a valuation ν over $\mathbf{x}$ where $\theta[\nu] \neq \lambda[\nu]$. The valuation ν serves as a *counterexample* to the equivalence query $EQ(\theta)$.

Consider the following examples. Assume $\lambda(x, y) = (x \wedge \neg y) \vee (\neg x \wedge y)$ is the target Boolean function over x and y. The teacher answers *NO* to the query $MEM(\nu)$ where $\nu(x) = \nu(y) = \mathsf{F}$ (denoted by $\nu(xy) = \mathsf{FF}$), since $\lambda(\mathsf{F}, \mathsf{F}) = \mathsf{F}$. For a different valuation $\nu(xy) = \mathsf{TF}$, the teacher answers *YES*. As an example of equivalence queries, consider $EQ(x \vee y)$. The teacher provides the valuation $\nu(xy) = \mathsf{TT}$ as a counterexample, since $\mathsf{T} \vee \mathsf{T} = \mathsf{T} \neq \mathsf{F} = \lambda(\mathsf{T}, \mathsf{T})$. For another equivalence query $EQ((x \vee \neg y) \wedge (\neg x \vee y))$, the teacher answers *YES*.

Let $\lambda(\mathbf{x})$ be a Boolean function over $\mathbf{x}$, $|\lambda(\mathbf{x})|_{DNF}$ and $|\lambda(\mathbf{x})|_{CNF}$ denote the sizes of $\lambda(\mathbf{x})$ in *minimal* disjunctive and conjunctive normal forms respectively. Under the aforementioned active learning model, the CDNF algorithm computes a representation for any target Boolean function $\lambda(\mathbf{x})$ with a polynomial number of queries in $|\lambda(\mathbf{x})|_{DNF}$, $|\lambda(\mathbf{x})|_{CNF}$, and $|\mathbf{x}|$ [4].

4 Learning a Contextual Assumption

Recall the following assume-guarantee reasoning rule (Lemma 1):

$$\frac{M_0 \| A \models \pi \qquad M_1 \preceq A}{M_0 \| M_1 \models \pi}$$

Our goal is to generate a contextual assumption $A = (\mathbf{x}_1, \iota_A(\mathbf{x}_1), \tau_A(\mathbf{x}_1, \mathbf{x}'_1))$ such that the premises $M_0 \| A \models \pi$ and $M_1 \preceq A$ hold. The contextual assumption consists of two parts: $\iota_A(\mathbf{x}_1)$ and $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$ which are Boolean functions over $\mathbf{x}_1$ and $\mathbf{x}_1 \cup \mathbf{x}'_1$ respectively. We naturally use the CDNF algorithm to learn both Boolean functions. Precisely, two instances of the CDNF algorithm are deployed: one for the initial predicate $\iota_A(\mathbf{x}_1)$, and the other for the transition relation $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$. Remember that the CDNF algorithm relies on a teacher, who knows the target Boolean function already, to answer queries from the learning algorithm. In this case, the target functions are unknown. We use the two premises of the assume-guarantee reasoning rule (Lemma 1) to simulate the role of a teacher. We explain in detail how this is done for the rest of Section 4.

There are four different types of queries (from the two instances of the CDNF algorithm) that need to be handled:

– the membership query $MEM(\mu)$ for the target $\iota_A(\mathbf{x}_1)$;
– the membership query $MEM(\mu, \mu')$ for the target $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$;
– the equivalence query $EQ(\iota)$ for the target $\iota_A(\mathbf{x}_1)$; and
– the equivalence query $EQ(\tau)$ for the target $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$.

In order to resolve membership queries, we exploit the fact that any contextual assumption must simulate M_1. The membership query $MEM(\mu)$ for the target $\iota_A(\mathbf{x}_1)$ is resolved by checking if μ satisfies $\iota_1(\mathbf{x}_1)$. If so, μ must also satisfy $\iota_A(\mathbf{x}_1)$ because M_1 is simulated by any contextual assumption A. The membership query $MEM(\mu, \mu')$ is resolved similarly.

For equivalence queries, we answer *YES* when a contextual assumption is found. Note that both conjectures $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ are needed to decide if they represent a contextual assumption. The two types of equivalence queries $EQ(\iota)$ and $EQ(\tau)$ hence cannot be resolved independently. In contrast to membership query resolution algorithms, there is only one equivalence query resolution algorithm for both types of equivalence queries.

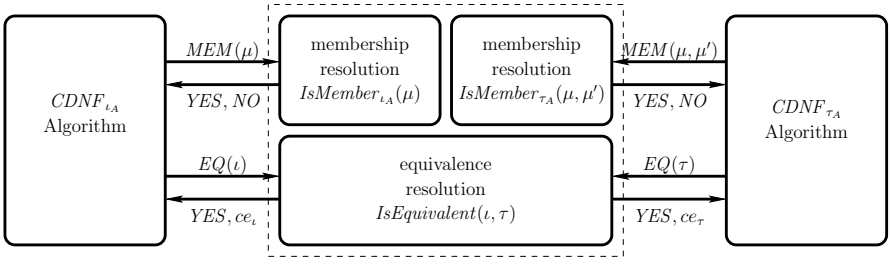


Fig. 1. Structure of Contextual Assumption Generator

Figure 1 shows the interaction between components in our contextual assumption generation algorithm. In the figure, two instances of the CDNF algorithm are shown on the sides. The instance $CDNF_{\iota_A}$ is intended to compute the initial predicate $\iota_A(\mathbf{x}_1)$ of an unknown contextual assumption A; the instance $CDNF_{\tau_A}$ is to compute the transition relation $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$ of A. The dashed box in the middle denotes the teachers. We design three query resolution algorithms to simulate the teachers for the two instances of the CDNF algorithm.

The membership query resolution algorithm $IsMember_{\iota_A}(\mu)$ resolves the membership query $MEM(\mu)$ for the target $\iota_A(\mathbf{x}_1)$. It receives queries and sends answers to the instance $CDNF_{\iota_A}$. Similarly, the membership query resolution algorithm $IsMember_{\tau_A}(\mu, \mu')$ communicates with the instance $CDNF_{\tau_A}$ solely. The equivalence query resolution algorithm $IsEquivalent(\iota, \tau)$, however, needs both conjectures from $CDNF_{\iota_A}$ and $CDNF_{\tau_A}$. It hence interacts with both instances.

4.1 Resolving Membership Queries

Let μ be a valuation over $\mathbf{x}_1$. The membership query $MEM(\mu)$ asks if μ is a satisfying valuation for the initial predicate $\iota_A(\mathbf{x}_1)$ of an unknown contextual

assumption A. We exploit the simulation relation in the assume-guarantee reasoning rule to resolve membership queries.

Input: $MEM(\mu)$: a membership query for the target $\iota_A(\mathbf{x}_1)$
Output: *YES* or *NO*
if $\iota_1[\mu] = \mathsf{T}$ **then return** *YES* **else return** *NO*;

(a) $IsMember_{\iota_A}(\mu)$

Input: $MEM(\mu, \mu')$: a membership query for the target $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$
Output: *YES* or *NO*
if $\tau_1[\mu, \mu'] = \mathsf{T}$ **then return** *YES* **else return** *NO*;

(b) $IsMember_{\tau_A}(\mu, \mu')$

Algorithm 1. Membership Query Resolution Algorithms

Algorithm 1a shows the membership query resolution algorithm for $MEM(\mu)$. In order to understand the algorithm, recall the premise $M_1 \preceq A$ in the assume-guarantee reasoning rule (Lemma 1). The initial predicate $\iota_A(\mathbf{x}_1)$ for any contextual assumption A must satisfy $\forall \mathbf{x}_1.\iota_1(\mathbf{x}_1) \implies \iota_A(\mathbf{x}_1)$. On the given valuation μ over $\mathbf{x}_1$, we hence check if $\iota_1[\mu] = \mathsf{T}$. If so, we have $\iota_A[\mu] = \mathsf{T}$ by $M_1 \preceq A$ and return *YES*. Otherwise, we simply return *NO* for the sake of termination. Observe that the answers to membership queries for the target $\iota_A(\mathbf{x}_1)$ are consistent with $\iota_1(\mathbf{x}_1)$. Algorithm 1a effectively targets the initial predicate $\iota_1(\mathbf{x}_1)$ of M_1. Subsequently, $CDNF_{\iota_A}$ can infer $\iota_1(\mathbf{x}_1)$ of M_1 as the initial predicate $\iota_A(\mathbf{x}_1)$ of an unknown contextual assumption eventually. Of course, one expects that an initial predicate different from $\iota_1(\mathbf{x}_1)$ will be learned. Our experiments show that this is indeed the case in practice.

Resolving membership queries $MEM(\mu, \mu')$ for the transition relation $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$ of an unknown contextual assumption is almost identical (Algorithm 1b). Let μ and μ' be valuations over $\mathbf{x}_1$ and $\mathbf{x}'_1$ respectively. Similar to the case of initial predicate, the transition relation $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$ of any contextual assumption A must satisfy $\forall \mathbf{x}_1, \mathbf{x}'_1.\tau_1(\mathbf{x}_1, \mathbf{x}'_1) \implies \tau_A(\mathbf{x}_1, \mathbf{x}'_1)$ due to $M_1 \preceq A$. If $\tau_1[\mu, \mu'] = \mathsf{T}$, $\tau_A[\mu, \mu'] = \mathsf{T}$ and hence our membership resolution algorithm returns *YES*. Otherwise, Algorithm 1b returns *NO*. As in the membership query resolution algorithm for the initial predicate, these answers make sure that $CDNF_{\tau_A}$ can infer the transition relation $\tau_1(\mathbf{x}_1, \mathbf{x}'_1)$ of M_1 and terminate eventually.

4.2 Resolving Equivalence Queries

Our equivalence query resolution algorithm answers two different types of equivalence queries from the two instances of the CDNF algorithm. The equivalence query $EQ(\iota)$ from $CDNF_{\iota_A}$ asks if the Boolean function $\iota(\mathbf{x}_1)$ represents the initial predicate of an unknown contextual assumption; $EQ(\tau)$ from $CDNF_{\tau_A}$ asks if the Boolean function $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ represents the transition relation of an unknown contextual assumption.

Let $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ be conjectures. Consider the transition system $C = (\mathbf{x}_1, \iota(\mathbf{x}_1), \tau(\mathbf{x}_1, \mathbf{x}'_1))$. Our equivalence query resolution algorithm first checks if M_1 is simulated by C. If M_1 is not simulated by C, the equivalence query resolution algorithm returns a counterexample to either $CDNF_{\iota_A}$ or $CDNF_{\tau_A}$. Otherwise, it continues to check if C is in fact a contextual assumption by verifying $M_0 \| C \models \pi$ with a Model Checking algorithm. If the composition of M_0 and C satisfies π, the equivalence query resolution algorithm returns *YES*. We conclude that $M_0 \| M_1$ satisfies π. If the composition of M_0 and C does not satisfy π, the equivalence query resolution algorithm examines the witness returned by the Model Checking algorithm. If the witness is also a witness to $M_0 \| M_1 \not\models \pi$, we conclude that $M_0 \| M_1$ does not satisfy π. Otherwise, the equivalence query resolution algorithm returns a counterexample to either $CDNF_{\iota_A}$ or $CDNF_{\tau_A}$.

Input: $EQ(\iota)$: an equivalence query for the target $\iota_A(\mathbf{x}_1)$; $EQ(\tau)$: an equivalence query for the target $\tau_A(\mathbf{x}_1, \mathbf{x}'_1)$
Output: *YES*, a counterexample to $EQ(\iota)$, or a counterexample to $EQ(\tau)$
let C be the transition system $(\mathbf{x}_1, \iota(\mathbf{x}_1), \tau(\mathbf{x}_1, \mathbf{x}'_1))$;
if $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ *is satisfied by* μ **then**
  answer $EQ(\iota)$ with the counterexample μ;
  receive another equivalence query $EQ(\iota')$;
  call *IsEquivalent*(ι', τ);
if $\tau_1(\mathbf{x}_1, \mathbf{x}'_1) \wedge \neg\tau(\mathbf{x}_1, \mathbf{x}'_1)$ *is satisfied by* $\mu\mu'$ **then**
  answer $EQ(\tau)$ with the counterexample $\mu\mu'$;
  receive another equivalence query $EQ(\tau')$;
  call *IsEquivalent*(ι, τ');
if $M_0 \| C \models \pi$ **then**
  answer $EQ(\iota)$ with *YES*;
  answer $EQ(\tau)$ with *YES*;
  report "$M_0 \| M_1 \models \pi$";
else
  let α be a witness to $M_0 \| C \not\models \pi$;
  call *IsWitness*(α);
end

Algorithm 2. *IsEquivalent*(ι, τ)

Algorithm 2 gives details of our equivalence query resolution algorithm. Let C be the transition system $(\mathbf{x}_1, \iota(\mathbf{x}_1), \tau(\mathbf{x}_1, \mathbf{x}'_1))$. To verify that M_1 is simulated by C, the algorithm checks if $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ is satisfiable. If $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ is satisfied by a valuation μ, then $\forall \mathbf{x}_1 . \iota_1(\mathbf{x}_1) \implies \iota(\mathbf{x}_1)$ does not hold and hence $M_1 \not\preceq C$. The valuation μ is returned to $CDNF_{\iota_A}$ as a counterexample to the equivalence query $EQ(\iota)$. The equivalence query resolution algorithm then restarts after it receives another conjecture from $CDNF_{\iota_A}$. Similarly, if $\tau_1(\mathbf{x}_1, \mathbf{x}'_1) \wedge \neg\tau(\mathbf{x}_1, \mathbf{x}'_1)$ is satisfied by $\mu\mu'$, the valuation $\mu\mu'$ is returned to $CDNF_{\tau_A}$ as a counterexample to the equivalence query $EQ(\tau)$.

Now assume $M_1 \preceq C$. That is, the second premise of the assume-guarantee reasoning rule is fulfilled. It remains to verify $M_0 \| C \models \pi$. The equivalence query resolution algorithm uses Model Checking to verify if $M_0 \| C \models \pi$. If $M_0 \| C \models \pi$,

both premises of the assume-guarantee reasoning rule are fulfilled. The equivalence resolution algorithm concludes $M_0 \| M_1 \models \pi$. Otherwise, the Model Checking algorithm returns a witness α to $M_0 \| C \not\models \pi$. Recall that M_1 is simulated by C and hence $Trace(M_1) \subseteq Trace(C)$. A witness α to $M_0 \| C \not\models \pi$ is not necessary a witness to $M_0 \| M_1 \not\models \pi$ for $\alpha{\downarrow_{\mathbf{x}_1}}$ may not be a trace of M_1. We therefore check whether $\alpha{\downarrow_{\mathbf{x}_1}} \in Trace(M_1)$ by the witness analysis algorithm.

Analyzing Witnesses. Given a witness α to $M_0 \| C \not\models \pi$, the witness analysis algorithm $IsWitness(\alpha)$ inspects α to see if $\alpha{\downarrow_{\mathbf{x}_1}}$ is also a trace of M_1. If so, α is a witness to $M_0 \| M_1 \not\models \pi$. Otherwise, the transition system C deviates from M_1 at some point in $\alpha{\downarrow_{\mathbf{x}_1}}$. The deviation is returned to either $CDNF_{\iota_A}$ or $CDNF_{\tau_A}$ as a counterexample to $EQ(\iota)$ or $EQ(\tau)$ respectively (Algorithm 3).

Input: α is a witness to $M_0 \| C \not\models \pi$
Output: a counterexample to $EQ(\iota)$, or a counterexample to $EQ(\tau)$
let $\alpha{\downarrow_{\mathbf{x}_1}} = \mu^0 \mu^1 \cdots \mu^t$;
if $\iota_1[\mu^0] = \mathsf{F}$ **then**
  answer $EQ(\iota)$ with the counterexample μ^0;
  receive another equivalence query $EQ(\iota')$;
  call $IsEquivalent(\iota', \tau)$;
for $i := 1$ **to** t **do**
  if $\tau_1[\mu^{i-1}, \mu^i] = \mathsf{F}$ **then**
    answer $EQ(\tau)$ with the counterexample $\mu^{i-1}\mu^i$;
    receive another equivalence query $EQ(\tau')$;
    call $IsEquivalent(\iota, \tau')$;
end
report "$M_0 \| M_1 \not\models \pi$ is witnessed by α";

Algorithm 3. $IsWitness(\alpha)$

More concretely, let $\alpha{\downarrow_{\mathbf{x}_1}} = \mu^0 \mu^1 \cdots \mu^t$ be a sequence of valuations over $\mathbf{x}_1$. Algorithm 3 verifies whether μ^0 is an initial state of M_1. If not, μ^0 is a counterexample to the equivalence query $EQ(\iota)$. Otherwise, the witness analysis algorithm checks if each transition of $\alpha{\downarrow_{\mathbf{x}_1}}$ on C is also a transition on M_1. If the i-th transition of $\alpha{\downarrow_{\mathbf{x}_1}}$ is not a transition on M_1 (that is, $\tau_1[\mu^{i-1}, \mu^i] = \mathsf{F}$), the valuation $\mu^{i-1}\mu^i$ is returned as a counterexample to the equivalence query $EQ(\tau)$. If a counterexample to either $EQ(\iota)$ or $EQ(\tau)$ is found, the equivalence query resolution algorithm waits for a new conjecture and then restarts. Otherwise, every transition of $\alpha{\downarrow_{\mathbf{x}_1}}$ is also a transition on M_1, α is a witness to $M_0 \| M_1 \not\models \pi$.

4.3 Correctness

The correctness of our assumption generation algorithm is established in three steps: proving soundness, completeness, and termination. Let $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}_i'))$ be transition systems for $i = 0, 1$ and $\pi(\mathbf{x})$ a state predicate over $\mathbf{x} = \mathbf{x}_0 \cup \mathbf{x}_1$. When the equivalence query resolution algorithm (Algorithm 2)

reports "$M_0 \| M_1 \models \pi$," it has verified that the composition of M_0 and C satisfies π, where $C = (\mathbf{x}_1, \iota(\mathbf{x}_1), \tau(\mathbf{x}_1, \mathbf{x}'_1))$ is the transition system corresponding to the conjectures $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$. Moreover, we have $M_0 \preceq C$ because both $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ and $\tau_1(\mathbf{x}_1, \mathbf{x}'_1) \wedge \neg\tau(\mathbf{x}_1, \mathbf{x}'_1)$ are not satisfiable. By the soundness of the assume-guarantee reasoning rule (Lemma 1), we have $M_0 \| M_1 \models \pi$.

On the other hand, when the witness analysis algorithm (Algorithm 3) reports "$M_0 \| M_1 \not\models \pi$ is witnessed by α," it has checked that $\alpha{\downarrow_{\mathbf{x}_1}}$ is a trace of M_1. Moreover, α is a witness to $M_0 \| C \not\models \pi$ and hence $\alpha{\downarrow_{\mathbf{x}_0}}$ is a trace of M_0. Since $\alpha{\downarrow_{\mathbf{x}_i}}$ is a trace of M_i for $i = 0, 1$, α is a trace of $M_0 \| M_1$ and thus a witness to $M_0 \| M_1 \not\models \pi$ as well. Our contextual assumption generation algorithm is sound.

Lemma 2 (soundness). *Let $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}'_i))$ be transition systems for $i = 0, 1$, and $\pi(\mathbf{x})$ a state predicate over $\mathbf{x} = \mathbf{x}_0 \cup \mathbf{x}_1$.*

1. *Let $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ be Boolean functions over $\mathbf{x}_1$ and $\mathbf{x}_1 \cup \mathbf{x}'_1$ respectively. If IsEquivalent(ι, τ) reports "$M_0 \| M_1 \models \pi$," then $M_0 \| M_1 \models \pi$;*
2. *Let $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ be Boolean functions over $\mathbf{x}_1$ and $\mathbf{x}_1 \cup \mathbf{x}'_1$ respectively. If IsEquivalent(ι, τ) reports "$M_0 \| M_1 \not\models \pi$ is witnessed by α," then α is a witness to $M_0 \| M_1 \not\models \pi$.*

If $M_0 \| M_1 \models \pi$, there is a transition system $C = (\mathbf{x}_1, \iota(\mathbf{x}_1), \tau(\mathbf{x}_1, \mathbf{x}'_1))$ such that $M_0 \| C \models \pi$ and $M_1 \preceq C$ by the invertibility of the assume-guarantee reasoning rule. Thus $\iota_1(\mathbf{x}_1) \wedge \neg\iota(\mathbf{x}_1)$ and $\tau_1(\mathbf{x}_1, \mathbf{x}'_1) \wedge \neg\tau(\mathbf{x}_1, \mathbf{x}'_1)$ are not satisfiable. Hence Algorithm 2 reports "$M_0 \| M_1 \models \pi$." On the other hand, assume α is a witness to $M_0 \| M_1 \not\models \pi$. Consider the transition system $C = (\mathbf{x}_1, \iota_{\mathsf{T}}(\mathbf{x}_1), \tau_{\mathsf{T}}(\mathbf{x}_1, \mathbf{x}'_1))$ where $\iota_{\mathsf{T}}(\mathbf{x}_1) = \mathsf{T}$ and $\tau_{\mathsf{T}}(\mathbf{x}_1, \mathbf{x}'_1) = \mathsf{T}$. Clearly $M_1 \preceq C$ and hence α is a witness to $M_0 \| C \not\models \pi$. Algorithm 3 reports "$M_0 \| M_1 \not\models \pi$ is witnessed by α." Our contextual assumption generation algorithm is complete.

Lemma 3 (completeness). *Let $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}'_i))$ be transition systems for $i = 0, 1$, and $\pi(\mathbf{x})$ a state predicate over $\mathbf{x} = \mathbf{x}_0 \cup \mathbf{x}_1$.*

1. *If $M_0 \| M_1 \models \pi$, then IsEquivalent(ι, τ) reports "$M_0 \| M_1 \models \pi$" for some Boolean functions $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ over $\mathbf{x}_1$ and $\mathbf{x}_1 \cup \mathbf{x}'_1$ respectively.*
2. *If α is a witness to $M_0 \| M_1 \not\models \pi$, then IsEquivalent$(\iota, \tau)$ reports "$M_0 \| M_1 \not\models \pi$ is witnessed by α" for some Boolean functions $\iota(\mathbf{x}_1)$ and $\tau(\mathbf{x}_1, \mathbf{x}'_1)$ over $\mathbf{x}_1$ and $\mathbf{x}_1 \cup \mathbf{x}'_1$ respectively.*

It remains to show that our algorithm always reports "$M_0 \| M_1 \models \pi$" or "$M_0 \| M_1 \not\models \pi$ is witnessed by α." Observe that the answers given by our query resolution algorithms are consistent with $\iota_1(\mathbf{x}_1)$ and $\tau_1(\mathbf{x}_1, \mathbf{x}'_1)$. Hence the instance $CDNF_{\iota_A}$ will infer $\iota_1(\mathbf{x}_1)$ after a polynomial number of queries. Similarly, $CDNF_{\tau_A}$ will generate $\tau_1(\mathbf{x}_1, \mathbf{x}'_1)$ eventually. At this point, the corresponding transition system $C = (\mathbf{x}_1, \iota_1(\mathbf{x}_1), \tau_1(\mathbf{x}_1, \mathbf{x}'_1)) = M_1$. The equivalence query resolution algorithm can always decide whether $M_0 \| M_1 \models \pi$ or not. Our contextual assumption generation algorithm therefore always reports to the user after a polynomial number of queries.

Lemma 4 (Termination). *Let* $M_i = (\mathbf{x}_i, \iota_i(\mathbf{x}_i), \tau_i(\mathbf{x}_i, \mathbf{x}_i'))$ *be transition systems for* $i = 0, 1$*, and* $\pi(\mathbf{x})$ *a state predicate over* $\mathbf{x} = \mathbf{x}_0 \cup \mathbf{x}_1$*. The contextual assumption generation algorithm reports "*$M_0 \| M_1 \models \pi$*" or "*$M_0 \| M_1 \not\models \pi$ *is witnessed by* α*" within a polynomial number of queries in* $|\iota_1(\mathbf{x}_1)|_{DNF}$*,* $|\iota_1(\mathbf{x}_1)|_{CNF}$*,* $|\tau_1(\mathbf{x}_1, \mathbf{x}_1')|_{DNF}$*,* $|\tau_1(\mathbf{x}_1, \mathbf{x}_1')|_{CNF}$*, and* $|\mathbf{x}_1|$*.*

5 Experiments

We have implemented a prototype of our contextual assumption generation algorithm in OCaml. Our current implementation uses the OCaml thread library for synchronization purposes. Each instance of the *CDNF* algorithm (that is, $CDNF_{\iota_A}$ or $CDNF_{\tau_A}$) is executed in a separate thread, and the equivalence query resolution algorithm is executed in a third thread.

We use MiniSat 2 (version 070721) in the membership query resolution algorithms (Algorithm 1) and the simulation checking in the equivalence query resolution algorithm (Algorithm 2). For monolithic Model Checking, we implement the interpolation-based algorithm in [19]. Interpolants are computed by instrumenting MiniSat 2. The interpolation-based Model Checking algorithm is also used in the equivalence query resolution algorithm (Algorithm 2).

We report three test cases in this section: the MSI cache coherence protocol [16], synchronous bus arbiters [18], and dining philosophers [22]. Each test case has experiments parametrized by the number of nodes. Let $M_1, \ldots, M_n$ be the nodes in an experiment with n nodes, and π a state predicate. We verify $M_1 \| \cdots \| M_n \models \pi$ in an experiment with n nodes.

Assume-guarantee reasoning is compared with monolithic interpolation-based Model Checking in each experiment. We explored several different partitions in each experiment. More precisely, an experiment with n nodes is divided into different partitions in n trials. In the i-th trial, we apply the following assume-guarantee reasoning rule:

$$\frac{(M_1 \| \cdots \| M_{i-1} \| M_{i+1} \| \cdots M_n) \| A \models \pi \qquad M_i \preceq A}{(M_1 \| \cdots \| M_{i-1} \| M_{i+1} \| \cdots M_n) \| M_i \models \pi}$$

Our contextual assumption algorithm generates a contextual assumption A to verify $M_1 \| \cdots \| M_n \models \pi$ in each trial. Since we do not address the decomposition problem in this paper, we choose the best result among the n trials and compare it with monolithic Model Checking. All experimental results are collected on a 3.2GHz Intel Xeon server with 2GB memory running Linux 2.4.20.

MSI Cache Coherence Protocol In the MSI cache coherence protocol, a memory is shared among n nodes. Each node has a cache. A bus connects the memory and caches of the nodes. When a node accesses a memory cell, it reads the cell from the bus and keeps a copy in its cache. Several copies of the same memory cell can be kept in different nodes. The MSI protocol ensures data coherence by keeping each cache in one of the three states: Modified, Shared, and Invalid [16]. Two properties are verified on the model derived from NuSMV [8]. We check that the first two nodes cannot own the bus simultaneously. Then we verify that

nodes	4	5	6	7	8	9	10	11	12
monolithic (sec)	2.6	4.1	4.9	5.3	6.0	7.9	7.6	9.3	9.6
assume-guarantee (sec)	1.5	1.9	4.0	2.7	3.6	6.3	7.1	7.6	8.6
improvement (%)	42.3	53.6	18.3	49.0	40.0	20.2	6.5	18.2	10.4
nodes	13	14	15	16	17	18	19	20	avg
monolithic (sec)	7.7	6.6	6.8	14.7	8.4	8.7	18.2	18.5	**8.6**
assume-guarantee (sec)	9.0	7.7	6.5	11.3	8.3	8.4	8.9	9.6	**6.6**
improvement (%)	-16.8	-16.6	4.4	23.1	1.1	3.4	51.0	48.1	**20.9**

(a) no contention for the first two nodes

nodes	4	5	6	7	8	9	10	11	12
monolithic	5s	15s	30s	42s	48s	1m43s	2m18s	5m8s	5m30s
assume-guarantee	3s	4s	30s	31s	31s	1m5s	42s	1m55s	1m33s
improvement (%)	40.0	73.3	0.0	26.1	35.4	36.8	69.5	62.6	71.8
nodes	13	14	15	16	17	18	19	20	avg
monolithic	2m37s	2m39s	3m14s	1m24s	6m38s	9m26s	9m26s	9m1s	**3m36s**
assume-guarantee	2m1s	2m20s	2m16s	1m28s	3m14s	4m5s	5m12s	9m11s	**2m9s**
improvement (%)	22.9	11.9	29.8	-4.7	51.2	56.7	44.8	-1.8	**36.8**

(b) no contention for all nodes

Fig. 2. Experimental Results for the MSI Protocol

any pair of nodes cannot own the bus at the same time. The former property involves only two nodes and is easier to verify than the latter. Figure 2 shows the results of experiments with 4 to 20 nodes.

In the figure, we show the verification time of the monolithic interpolation-based Model Checking (*monolithic*), the verification time of assume-guarantee reasoning (*assume-guarantee*), and the ratio of improvement (*improvement*). On the first property, monolithic Model Checking takes more than 14 seconds in the experiments with 16, 19, and 20 nodes. Assume-guarantee reasoning, on the other hand, finishes all but one experiments in 10 seconds. Assume-guarantee reasoning also performs significantly better than monolithic Model Checking on the second property. The verification time for assume-guarantee reasoning increases more stably than monolithic Model Checking (Figure 2b). The generated contextual assumptions improve assume-guarantee reasoning by 50% in 5 experiments with no less than 10 nodes. Given an experiment in this test case, one expects assume-guarantee reasoning to outperform monolithic Model Checking by 20.9% and 36.8% on the two properties respectively.

Synchronous Bus Arbiters. The synchronous bus arbiter is a bus arbitration protocol for synchronous circuits [18]. In this protocol, n nodes are connected in a ring. A token is passed around the nodes. A node can request and acknowledge the token from the node next to it. The node having the token has the exclusive right to access the bus. We generalize the model in NuSMV [8] and verify two properties in this test case. We check that the first pair of nodes cannot

nodes	4	5	6	7	8	9	10	11	12
monolithic (sec)	5.1	7.6	11.1	16.6	25.5	42.4	58.9	81.1	123.7
assume-guarantee (sec)	4.2	6.4	10.5	14.5	22.9	36.4	41.3	45.8	108.2
improvement (%)	17.6	15.7	5.4	12.6	10.1	14.1	29.8	43.5	12.6
nodes	13	14	15	16	17	18	19	20	avg
monolithic (sec)	159.3	130.6	314.0	81.3	423.1	548.8	698.3	900.0	**213.3**
assume-guarantee (sec)	139.6	115.0	188.9	61.1	374.4	463.3	531.9	568.2	**160.7**
improvement (%)	12.3	11.9	39.8	24.8	11.5	15.5	23.8	36.8	**19.8**

(a) no simultaneous acknowledgment for the first two nodes

nodes	4	5	6	7	8	9	10	11	12
monolithic	3s	5s	5s	10s	34s	34s	1m45s	1m51s	4m32s
assume-guarantee	3s	5s	5s	10s	34s	50s	1m44s	1m59s	4m33s
improvement (%)	0	0	0	0	0	-47.0	0.9	-7.2	-0.3
nodes	13	14	15	16	17	18	19	20	avg
monolithic	7m9s	10m54s	12m27s	21m2s	30m22s	24m3s	33m38s	45m29s	**11m35s**
assume-guarantee	7m4s	8m43s	8m43s	12m39s	17m57s	24m0s	33m22s	45m20s	**9m43s**
improvement (%)	1.1	20.0	29.9	39.8	40.8	0.2	0.7	0.3	**4.6**

(b) no simultaneous acknowledgment for any pair of nodes

Fig. 3. Experimental Results for Synchronous Bus Arbiters

acknowledge the token simultaneously. Then we check that any pair of nodes cannot acknowledge the token at the same time. Figure 3 shows the results.

For the first property, assume-guarantee reasoning outperforms monolithic Model Checking consistently. Our algorithm computes a contextual assumption that improves the verification time by 19.8% on average. Assume-guarantee reasoning decisively outperforms monolithic Model Checking for experiments with 14 to 17 nodes on the second property. Among the experiments in all three cases, the experiments with 9 nodes is the only one where assume-guarantee reasoning is outperformed by more than 20%. Subsequently, assume-guarantee reasoning does not significantly improve the verification time on this property (4.6%).

Dining Philosophers. The dining philosophers problem illustrates a simple resource sharing problem in concurrent programs. In dining philosophers, n nodes are connected in a ring. Neighboring nodes share a resource. A node requires both resources shared with its neighbors to enter its working mode [22]. In this test case, we verify that a fixed pair of neighboring nodes cannot enter their working modes simultaneously (Figure 4).[1]

Our experiments show that the verification time of monolithic Model Checking varies drastically in this case. Assume-guarantee reasoning, on the other hand, performs more stably. Take the experiment with node 17 as an example.

[1] In fact, verifying that any neighboring nodes cannot enter the working mode in dining philosophers takes so much time that both monolithic Model Checking and assume-guarantee reasoning cannot finish in one hour in a setting with only 4 philosophers.

nodes	4	5	6	7	8	9	10	11	12
monolithic (sec)	15.8	16.6	823.7	141.1	22.7	56.1	32.0	34.7	64.3
assume-guarantee (sec)	13.1	11.3	33.3	15.1	10.9	19.6	32.2	23.6	32.1
improvement (%)	17.0	21.0	95.9	89.2	51.9	65.0	-0.6	31.9	50.0
nodes	13	14	15	16	17	18	19	20	avg
monolithic (sec)	1109.9	60.6	46.1	32.7	1741.1	91.1	2406.7	63.7	**397.5**
assume-guarantee (sec)	29.5	34.3	36.8	28.9	58.8	66.4	39.5	67.5	**32.5**
improvement (%)	97.3	43.3	20.1	11.6	96.6	27.1	98.3	-5.9	**47.6**

Fig. 4. Experimental Results for Dining Philosophers

Interpolation-based algorithm uses 180MB memory to compute 8 interpolants to conclude that the property is verified. Assume-guarantee reasoning only uses 104MB memory and 7 interpolants to reach the same conclusion. With our contextual assumption generation algorithm, assume-guarantee reasoning is expected to outperform monolithic Model Checking by 47.6% in this test case.

6 Conclusion

We introduced a new contextual assumption generation algorithm in this paper. The new algorithm computes implicit representations and is more scalable than explicit automata-theoretic algorithms. With the contextual assumptions generated by our algorithm, assume-guarantee reasoning can improve monolithic interpolation-based Model Checking in three parametrized test cases.

The initial predicate and the transition relation of the generated contextual assumption are different from those of a node. In all 1020 $(= (2+2+1) \times \sum_{n=4}^{20} n)$ trials, each generated contextual assumptions has different initial predicates and transition relations from those of its target node. Moreover, since the generated contextual assumption simulates its target, it is in fact an abstraction of the target node [9,3]. Although our contextual assumption generation algorithm can apply abstraction refinement techniques implemented in Model Checkers, it will be interesting to compare these two techniques.

Targeting one node is not the best decomposition we have in our test cases. In the MSI cache coherence protocol, targeting all nodes allows assume-guarantee reasoning to verify the experiment with 36 nodes in 4 minutes whereas monolithic Model Checking uses up all memory in 9 minutes and fails to verify. The challenge of how to best decompose a problem still remains. In summary, our experiments show that there is always *a decomposition* to make assume-guarantee reasoning outperform monolithic interpolation-based Model Checking in the three test cases. Finding such a decomposition will certainly be an important future work.

References

1. Angluin, D.: Learning regular sets from queries and counterexamples. Information and Computation 75(2), 87–106 (1987)
2. Barringer, H., Giannakopoulou, D., Păsăreanu, C.S.: Proof rules for automated compositional verification through learning. In: Workshop on Specification and Verification of Component-Based Systems, pp. 14–21 (2003)
3. Bobaru, M.G., Păsăreanu, C.S., Giannakopoulou, D.: Automated assume-guarantee reasoning by abstraction refinement. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 135–148. Springer, Heidelberg (2008)
4. Bshouty, N.H.: Exact learning boolean function via the monotone theory. Information and Computation 123(1), 146–153 (1995)
5. Chaki, S., Clarke, E.M., Sinha, N., Thati, P.: Automated assume-guarantee reasoning for simulation conformance. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 534–547. Springer, Heidelberg (2005)
6. Chaki, S., Strichman, O.: Optimized L^*-based assume-guarantee reasoning. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 276–291. Springer, Heidelberg (2007)
7. Chen, Y.F., Farzan, A., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Learning minimal separating DFA's for compositional verification. In: Kowalewski, S., Philippou, A. (eds.) TACAS 2009. LNCS, vol. 5505, pp. 31–45. Springer, Heidelberg (2009)
8. Cimatti, A., Clarke, E., Giunchiglia, F., Roveri, M.: NuSMV: a new Symbolic Model Verifier. In: Halbwachs, N., Peled, D. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
9. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM 50(5), 752–794 (2003)
10. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: An evaluation of automated assume-guarantee reasoning. ACM Trans. Software Engineering Methodology 17(2) (2008)
11. 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)
12. Farzan, A., Chen, Y.F., Clarke, E.M., Tsay, Y.K., Wang, B.Y.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
13. Gheorghiu, M., Giannakopoulou, D., Păsăreanu, C.S.: Refining interface alphabets for compositional verification. In: Grumberg, O., Huth, M. (eds.) TACAS 2007. LNCS, vol. 4424, pp. 292–307. Springer, Heidelberg (2007)
14. Giannakopoulou, D., Păsăreanu, C.S.: Special issue on learning techniques for compositional reasoning. Formal Methods in System Design 32(3), 173–174 (2008)
15. Gupta, A., McMillan, K.L., Fu, Z.: Automated assumption generation for compositional verification. Formal Methods in System Design 32(3), 285–301 (2008)
16. Handy, J.: The Cache Memory Book. Academic Press, London (1998)
17. Jung, Y., Kong, S., Wang, B.Y., Yi, K.: Deriving invariants in propositional logic by algorithmic learning, decision procedure, and predicate abstraction. In: VMCAI. LNCS. Springer, Heidelberg (2010)
18. McMillan, K.L.: The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University (1992)

19. McMillan, K.L.: Interpolation and SAT-based model checking. In: Hunt Jr., W.A.H., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 1–13. Springer, Heidelberg (2003)
20. Nam, W., Madhusudan, P., Alur, R.: Automatic symbolic compositional verification by learning assumptions. Formal Methods in System Design 32(3), 207–234 (2008)
21. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Information and Computation 103(2), 299–347 (1993)
22. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts, 7th edn. John Wiley & Sons, Inc., Chichester (2004)
23. Sinha, N., Clarke, E.M.: SAT-based compositional verification using lazy learning. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 39–54. Springer, Heidelberg (2007)

Learning Component Interfaces with May and Must Abstractions

Rishabh Singh[1,⋆], Dimitra Giannakopoulou[2], and Corina Păsăreanu[2]

[1] MIT CSAIL/ MCT Inc.
[2] CMU/ NASA Ames

Abstract. Component interfaces are the essence of modular program analysis. In this work, a component interface documents correct sequences of invocations to the component's public methods. We present an automated framework that extracts finite *safe*, *permissive*, and *minimal* interfaces, from potentially infinite software components. Our proposed framework uses the L* automata-learning algorithm to learn finite interfaces for an infinite-state component. It is based on the observation that an interface *permissive* with respect to the component's must abstraction and *safe* with respect to its may abstraction provides a precise characterization of the legal invocations to the methods of the concrete component. The abstractions are refined automatically from counterexamples obtained during the reachability checks performed by our framework. The use of must abstractions enables us to avoid an exponentially expensive determinization step that is required when working with may abstractions only, and the use of L* guarantees minimality of the generated interface. We have implemented the algorithm in the ARMC tool and report on its application to a number of case studies including several Java2SDK and J2SEE library classes as well as to NASA flight-software components.

1 Introduction

Component interfaces are a central concept in component-based software engineering. In current practice, interfaces typically describe the services that a component *provides* and *requires* at a purely syntactic level. However, the need has been identified for interfaces that document richer aspects of component behavior. For example in this work, as in others [1,5,8,11,12,16], interfaces describe correct sequences of invocations to public methods of a component. Richer interfaces can serve as a documentation aid to application programmers, but can also be used by verification tools in checking that the components are invoked correctly within a system. In fact, interfaces are key for modular program analysis [8,11,12]. They reduce the task of verifying a system consisting of a component and a client, to the more tractable task of verifying that the client satisfies the component's interface.

⋆ Work done while the author, supported by MCT Inc., was visiting NASA Ames.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 527–542, 2010.

Given the source-code of a library component C, we address the problem of extracting a precise component interface in the form of a deterministic finite-state automaton (DFA)[1], labeled with the public method names of C. By precise, we mean *safe* and *permissive*. An interface is safe if it accepts no illegal sequence of calls to C, and permissive if it includes all the legal sequences of calls to C [16]. In contrast to our previous work [12], we combine interface generation algorithms with predicate abstraction techniques, that allows us to handle components with very large or infinite state spaces. The novelty of our proposed algorithms lies in the fact that we use a combination of under-, and over-approximations of the component behavior, in the form of must and may abstractions, respectively. Our approach is based on the observation that an interface that is safe with respect to the may abstraction and permissive with respective to the must abstraction is safe and permissive with respect to C itself. We use the L* learning algorithm [4] to generate safe and permissive interfaces for C, by iteratively checking may and must abstractions of C. These abstractions are gradually refined during the learning process, based on counterexamples. If the algorithms terminate, then the returned interface is the *minimal* DFA capturing the precise interface for C.

Extended interfaces can be difficult to characterize precisely without the help of automated tools, making interface generation an area of active research [1,5,16]. The approaches closest to ours are those presented in [1,16]. Both approaches construct only over-approximations of the component behavior, which may be non-deterministic. Checking permissiveness when (abstracted) components are non-deterministic requires a potentially expensive determinization step. Alur et al. [1] avoid this step by using heuristics, and therefore cannot guarantee permissiveness of the generated interfaces. On the other hand, Henzinger et al. [16] build "abstract regions", which is equivalent to performing a determinization step. Their abstractions are subsequently checked for safety and permissiveness. These steps cannot be combined in an on-the-fly algorithm, so the complete abstract reachability graph needs to be constructed, even if a counterexample exists early in the search.

Furthermore, the abstraction mechanisms in [16] cannot guarantee minimal interfaces. Even if these interfaces were to be minimized, this approach would suffer from potentially large intermediate interfaces that subsequently get compacted. This latter problem is more pronounced in the presence of the determinization step, which is exponential, in the worst case. In contrast, L*-based approaches like ours and [1] directly generate minimal interfaces. Note however that the technique by [1] does not provide criteria to automatically detect the need for abstraction refinement. Their refinements are based on inspection of the generated interfaces, and are performed manually.

Contributions. We present a framework for automated generation of *minimal*, *safe* and *permissive* interfaces for large or infinite-state components. The framework uses L* with automatically generated and refined may and must abstractions of the component behavior. It guarantees permissiveness without

[1] In this work, we assume that component interfaces have a regular language. We therefore do not consider components methods with recursive invocations.

requiring determinization, and performs all checks on-the-fly. We present a basic algorithm and also an optimized version that re-uses results across abstraction-refinement iterations. We also describe the implementation of our algorithm in ARMC, and the application to the benchmarks presented in [1,16], as well as new benchmarks including J2SEE classes and NASA software components.

Other related work. Work on predicate abstraction for modal transition systems, e.g. [13], similarly distinguishes between may and must transitions. However, to the best of our knowledge, the use of may and must abstractions for interface generation is novel. Other approaches generate interfaces by using static analysis [27], or a combination of static and dynamic analyses [28], or by extracting information from sample execution traces [3]. All these techniques generate approximate interfaces, as opposed to our work that aims at producing precise interfaces that provide correctness guarantees.

Interface generation is related to assume-guarantee reasoning [2,10,18,23], since component interfaces can be used as assumptions in this context. Shoham et al. [26] describe a compositional framework for modal transition systems, based on techniques taken from the 3-valued game-based model checking for abstract models [9,14]. Those approaches do not use explicit interfaces (or assumptions). Finally, recent work [15] uses may and must information in the form of procedure summaries in a compositional framework that performs program analysis.

2 Example

Our running example, taken from [16], is illustrated in Figure 1. It consists of a component C with 3 static variables and 6 public library methods. Variable e defines the error states in the component ($e \neq 0$), variable a denotes the possession of lock and variable x enables method write. Methods acq/rel and their variations acqx/relx are used to acquire/release a lock, respectively. Methods read and write are used to access and update the shared memory, respectively.

It can be observed that C enforces several requirements such as read can only be called after acq or acqx. Similarly write can be called safely only after calling acqx. Once acq is called, it can only be called again after calling rel or relx. The interface A for C should capture all such correct sequences of invocation of public methods and reject the incorrect ones.

```
void rel(){
  a = NULL;
  return;}

void relx(){
  a = NULL;
  x = 0;
  return;}
```

```
void acq(){
 if(a==NULL)
    a=get_lock();
 else
    e=1;
 return;}
```

```
void read(){
  if(a!=NULL)
     m_read(a);
 else
    e=1;
 return;}
```

```
void acqx(){
 if(a==NULL){
    a=get_lock();
    x=1;}
 else
    e=1;
 return;}
```

```
void write(){
 if(x!=0)
     m_write(a);
 else
    e=1;
 return;}
```

Fig. 1. Read-write-acq example

3 Preliminaries

Components and Interfaces. A component $C = (X_s, F, s_0, P_{err}, \Sigma)$ consists of: a set X_s of static global variables shared across the methods ($[[X_s]]$ denotes the valuations of variables in X_s and represents the states of the component); a set F of *library methods*; initial state s_0 of the component, $s_0 \in [[X]]$; a global set P_{err} of error predicates over variables in X; and a finite alphabet Σ of the method names. The error predicates denote the error conditions in the library such as runtime exceptions, assertion violations etc. A component state $s \in [[X_s]]$ is an *error state* if s satisfies an error predicate.

Example 1. The example component in Figure 1 can be expressed as $C = (X, F, s_0, P_{err}, \Sigma)$ where $X = \{a, x, e\}$, F is the set of CFAs for methods (described below), the start state $s_0 = \{a = NULL, x = 0, e = 0\}$, the error predicate $P_{err} = \{e \neq 0\}$ and alphabet set $\Sigma = \{\mathsf{acq}, \mathsf{read}, \mathsf{rel}, \mathsf{write}, \mathsf{relx}, \mathsf{acqx}\}$.

Every library method $f \in F$ is represented as a *control-flow automaton* (CFA) $f = (X_s, X_l, Q, q_s, q_r, \mathcal{T})$ consisting of a disjoint set of static variables (X_s) and local variables (X_l), a set Q of *control locations*; a start location $q_s \in Q$, a return location $q_r \in Q$, and a finite set of *method transitions* $\mathcal{T}$. Each transition $\tau \in \mathcal{T}$ is labeled with a *from* location $q_{from} \in Q$, a *to* location $q_{to} \in Q$ and the method statement operation represented as a guarded command, $g(\boldsymbol{x}) \mapsto \boldsymbol{x} = e(\boldsymbol{x})$ where $g(\boldsymbol{x})$ is a guard and $e(\boldsymbol{x})$ are updates to variables in $\boldsymbol{x} \in (X_s \cup X_l)$. We use a special no-op skip transition to model multiple return locations with one return location.

CFA and Component Semantics. We give the definition of CFA semantics in terms of method transitions and of component semantics in terms of method calls. A state in the CFA is modelled as (q, s) where $q \in Q$ is a control location and $s \in [[(X_s \cup X_l]]$ represents the valuation of (both global and local) variables in that state, whereas a state in the component is represented by s where $s \in [[X_s]]$ denotes the valuation of (only global) variables in that state.

A binary transition relation $\rho_\tau \subseteq (Q \times [[X_s \cup X_l]])^2$ captures the semantics of the transitions $\tau \in \mathcal{T}$ in a CFA. $((q, s), (q', s')) \in \rho_\tau$ if $q = \tau.q_{from}$, $q' = \tau.q_{to}$, $s \models \tau.g$ and $s' = \tau.e(s)$. We write $s \xrightarrow{\tau} s'$ for $((\tau.q_{from}, s), (\tau.q_{to}, s')) \in \rho_\tau$.

Let $s \circ t$ denote the combination of valuations $s \in [[X_s]]$ (static variables) and $t \in [[X_l]]$ (local variables). The transition relation for the component $\delta_C \subseteq [[X_s]] \times \Sigma \times [[X_s]]$ denotes the successful termination of method f when applied on some state $s \in [[X_s]]$ resulting in state $s' \in [[X_s]]$. It is defined as follows: $(s, f, s') \in \delta_C$ if $\exists$ sequence $(q_1, (s_1 \circ t_1)), (q_2, (s_2 \circ t_2)), \ldots, (q_n, (s_n \circ t_n))$ such that $(q_s, s) = (q_1, s_1)$ (q_1 is the start location of f), $(q_r, s') = (q_n, s_n)$ (q_n is the return location of f), and $\underset{1 \leq i \leq n-1}{\forall i} ((q_i, (s_i \circ t_i)), (q_{i+1}, (s_{i+1} \circ t_{i+1}))) \in f_c.\rho_\tau, s_i \in [[X_s]], t_i \in [[X_l]]$ (every transition in the sequence is a valid transition in $f.\mathcal{T}$). For simplicity we assume error states have no outgoing method transitions, except for return. We write $s \xrightarrow{f} s'$ for $(s, f, s') \in \delta_C$.

The semantics of the component C is captured by a (possibly infinite) deterministic transition system $S_C = ([[X]], \Sigma, s_0, \delta_C)$. A *component sequence*

$\mathsf{Seq} = f_1, f_2, \ldots, f_n$ is the sequence of method calls corresponding to a computation $s_0, s_1, \ldots, s_n$ of S_C such that $\forall i\ s_i \in [[X]], (s_{i-1}, f_i, s_i) \in \delta_C$. An error sequence is a component sequence that leads the component to an error state. The language $L(S_C) \subseteq \Sigma^*$ denotes all the component sequences of C; $L^E(S_C) \subseteq L(S_C)$ denotes the language of error sequences, and $L^{\mathsf{safe}}(S_C)$ denotes the language of *safe method sequences* which is defined to be the complement of $L^E(S_C)$, i.e. $L^{\mathsf{safe}}(S_C) = \overline{L^E(S_C)}$. Note that while $L(S_C)$ and $L^E(S_C)$ contain only *feasible* traces, $L^{\mathsf{safe}}(S_C)$ may contain both feasible and infeasible component sequences.

Safe and Permissive Interfaces. An interface for a library component C is a prefix-closed regular set over the alphabet Σ of library method names. We represent interfaces as (deterministic) finite state automata $A = (Q, \Sigma, q_0, \delta)$ where: Q is a finite non-empty set of accept states; Σ is a finite alphabet of method names; $q_0 \in Q$ is the initial state; and the transition relation $\delta \subseteq Q \times \Sigma \times Q$ (the set of accepting states is Q). $L(A)$ is the set of words accepted by A. We let $L^E(A) = \overline{L(A)}$ denote the set of error traces of A. $L^E(A)$ is the language accepted by automaton A_{err}, representing A completed with an error state which is the only accepting state, i.e., $A_{\mathsf{err}} = (Q', \Sigma, q_0, \delta')$, where $Q' = Q \cup \{\mathsf{err}\}$ and $\delta' = \delta \cup (q, a, \mathsf{err})\ \forall q, q' \in Q, a \in \Sigma : (q, a, q') \notin \delta$.

Interface A is safe if *every word* $w \in L(A)$ is a safe sequence of method calls in C, i.e. $L(A) \subseteq L^{\mathsf{safe}}(S_C)$; equivalent to $L^E(S_C) \subseteq L^E(A)$ or $L(A) \cap L^E(S_C) = \emptyset$.

Interface A is permissive if it accepts *all* safe sequences of method calls in C, i.e. $L^{\mathsf{safe}}(S_C) \subseteq L(A)$; equivalent to $L^E(A) \subseteq L^E(S_C)$ or $L^{\mathsf{safe}}(S_C) \cap L^E(A) = \emptyset$.

From the above definitions, since $L^E(S_C) \subseteq L^E(A)$ and $L^E(A) \subseteq L^E(S_C)$, it follows that $L^E(S_C) = L^E(A)$.

Example 2. For the component in Figure 1, the string $\sigma_1 = (\mathsf{acq},\mathsf{read},\mathsf{rel}) \in L^{\mathsf{safe}}(S_C)$ and $\sigma_1 \in L(S_C)$ as the corresponding method sequence is safe for the component. The string $\sigma_2 = (\mathsf{read},\mathsf{acq},\mathsf{rel}) \in L^E(S_C)$ as the method sequence causes the component to be in an error state.

Composition. Let $S_C = ([[X]], \Sigma, s_0, \delta_C)$ be the transition system capturing the semantics of library component C, and $A = (Q, \Sigma, q_0, \delta)$ be an interface automaton. The composite transition system $G = S_C \parallel A$ obtained by composing S_C and A is defined as $G = (Q^\times, \Sigma, q_0^\times, \delta^\times)$, where $Q^\times = Q \times [[X]]$, $q_0^\times = (q_0, s_0)$, and $\delta^\times = \{((q, s), f, (q', s'))|(q, f, q') \in \delta \text{ and } (s, f, s') \in \delta_C\}$.

Abstraction. We build *may* and *must* abstractions of software components using predicate abstraction – a special instance of abstract interpretation [7] that maps a potentially infinite state transition system into a finite state transition system via a finite set of predicates $\mathsf{Preds} = \{p_1, \ldots, p_n\}$ over the program variables. We require $P_{err} \subseteq \mathsf{Preds}$. An abstract state $a \subseteq \mathsf{Preds}$ is an *error state* if it satisfies an error predicate.

An abstraction function α maps a concrete program state s to a set of predicates that hold in s: $\alpha(s) = \{p \in \mathsf{Preds} \mid s \models p\}$. For transition $\tau \in \mathcal{T}$ of method f, we define *may* and *must* transitions; a, a' denote abstract states, s, s' denote concrete states:

- $a \xrightarrow{\tau}_{must} a'$ iff $\forall s$ s.t. $\alpha(s) = a$, $\exists s'$ s.t. $\alpha(s') = a'$ and $s \xrightarrow{\tau} s'$.
  $a \xrightarrow{f}_{must} a'$ iff $\forall s$ s.t. $\alpha(s) = a$, $\exists s'$ s.t. $\alpha(s') = a'$ and $s \xrightarrow{f} s'$.
- $a \xrightarrow{\tau}_{may} a'$ iff $\exists s$ s.t. $\alpha(s) = a$ and $\exists s'$ s.t. $\alpha(s') = a'$ and $s \xrightarrow{\tau} s'$.
  $a \xrightarrow{f}_{may} a'$ iff $\exists s$ s.t. $\alpha(s) = a$ and $\exists s'$ s.t. $\alpha(s') = a'$ and $s \xrightarrow{f} s'$.

Given component C with transition system S_C, the must and may abstractions with respect to the set of abstract predicates Preds are defined as $S^{must}_{C,\mathsf{Preds}} = (2^{\mathsf{Preds}}, \Sigma, \alpha(s_0), \longrightarrow_{must})$ and $S^{may}_{C,\mathsf{Preds}} = (2^{\mathsf{Preds}}, \Sigma, \alpha(s_0), \longrightarrow_{may})$, respectively. We sometimes write S^{must}_C or S^{may}_C when Preds is clear from the context.

Algorithms for computing may and must abstractions with the help of a theorem prover are given in e.g. [24]. Note that the set of may transitions is a super-set of the must transitions. We also note from the above definitions it follows that the may and must abstractions define simulations [21] between S^{must}_C and S_C, and between S_C and S^{may}_C, respectively. Since simulation implies trace inclusion, we have the following characterization of under- and over- approximations (that we will use later):

Proposition 1. $L(S^{must}_C) \subseteq L(S_C) \subseteq L(S^{may}_C)$. *Furthermore,* $L^E(S^{must}_C) \subseteq L^E(S_C) \subseteq L^E(S^{may}_C)$.

Weakest Precondition. For automated abstraction refinement, we use weakest precondition calculations over counterexample traces [24]. Let ϕ be a predicate characterizing a set of states. The weakest precondition of ϕ with respect to transition τ is $\mathsf{wp}(\phi, \tau) = \forall s'.(s \xrightarrow{\tau} s' \Rightarrow \phi(s'))$, and it characterizes the largest set of states whose successors by transition τ satisfy ϕ.

The L^* Algorithm. L* was developed by Angluin [4] and later improved by Rivest and Schapire [25]. L* learns an unknown regular language U over alphabet Σ and produces a *minimal deterministic* finite state automaton (DFA) that accepts it. L* interacts with a *Minimally Adequate Teacher* that answers two types of questions from L*. The first type is a *membership query* asking whether a string $\sigma \in \Sigma^*$ is in U. For the second type, the learning algorithm generates a *conjecture* A and asks whether $L(A) = U$. If $L(A) \neq U$ the Teacher returns a counterexample, which is a string σ in the symmetric difference of $L(A)$ and U. L* is guaranteed to terminate with a minimal automaton A for U. If A has n states, L* makes at most $n - 1$ incorrect conjectures. The number of membership queries made by L* is $O(kn^2 + n \log m)$, where k is the size of Σ, n is the number of states in the minimal DFA for U, and m is the length of the longest counterexample returned when a conjecture is made.

4 Interface Generation

Let C be a component corresponding to a potentially infinite-state transition system S_C. From now on, for simplicity, we will use C to represent the component and its transition system. Our proposed interface-generation algorithms operate by analyzing *finite-state* abstractions of C. The essence of our approach lies in the following observation:

Theorem 1. *Let us assume a component C, a may abstraction C^{may} for C and a must abstraction C^{must} for C. If an interface A for C is* permissive *with respect to C^{must} and* safe *with respect to C^{may}, then A is* safe *and* permissive *with respect to C.*

Our approach for interface generation is therefore based on constructing may and must abstractions for a concrete component C (C^{may} and C^{must}, respectively). We first briefly describe a basic algorithm, followed by an optimized one; both algorithms use a combination of automated learning and abstraction refinement techniques. These algorithms involve procedures for checking whether an interface is safe and permissive, which we provide first.

`CheckSafe.` Checking that an interface A is safe for some component abstraction C^{Abst} (corresponding to C^{may} or C^{must}), reduces to checking reachability of a state (s_a, s_c) in $A \parallel C^{\mathsf{Abst}}$ such that s_c is an error state in C^{Abst}. A counterexample is returned if such a state is found.

`CheckPermissive.` The key to our approach is that our algorithms only check permissiveness for C^{must}. Must abstractions are always deterministic since we assume that our concrete components are also deterministic. As a result, checking permissiveness reduces to a simple reachability check. The interface A is first completed with an error state err to get A_{err}. C^{must}_{sink} is then built by similarly completing C^{must} with a new state sink, which is an accepting state (see [12] for explanations of the need for such completions). The permissiveness check then reduces to checking, in automaton $A_{\mathsf{err}} \parallel C^{must}_{sink}$, reachability of some state (err, s_c), where s_c is a non-error state in C^{must}_{sink}. If such a state is detected, A is not permissive, and a counterexample is returned. The counterexample illustrates a correct sequence of invocations to the component that is rejected by the interface.

4.1 Algorithms

Algorithm BuildInterface**:** The high-level steps of our basic approach to generating interfaces using may and must abstractions is illustrated in Algorithm 1. Given that C^{must} is finite-state, the L* algorithm is used to generate a safe and permissive interface for C^{must}, expressed as a DFA A^{must} over the alphabet of the component. The procedure LearnInterface used for this purpose is similar to the one presented in [12]. The interface A^{must} produced by LearnInterface is subsequently checked for safety with respect to C^{may}. If safe, then based on theorem 1, A^{must} is a safe and permissive interface for C. Otherwise, the counterexample t obtained from the safety check is used to guide the automatic refinement of the predicate set used for building the component abstractions, as described later in this section. Another iteration of the algorithm is then performed, with the new set of predicates.

Algorithm LearnReuse**:** Despite its conceptual clarity, the basic algorithm needs to restart the learning process every time an abstraction is refined. We would ideally like to reuse information learned by L* across abstraction refinement iterations. In contrast to the basic algorithm that uses learning on C^{must}, the

Algorithm 1. BuildInterface(C)

```
1: A^must := LearnInterface(C^must )
2: t := CheckSafe(A^must ,C^may )
3: if t == null then
4:    return A^must
5: else
6:    Preds := Preds ⋃ Refine(t)
7:    Go to step 1.
8: end if
```

Algorithm 2. Query(σ, C)

```
1: if CheckSafe(ts(σ), C^must )! = null
   then
2:    return no
3: else
4:    t := CheckSafe(ts(σ),C^may )
5:    if t == null then
6:       return yes
7:    else
8:       Preds := Preds ⋃ Refine(t)
9:       invoke Query(σ, C) (new Preds)
10:   end if
11: end if
```

Algorithm 3. Oracle 1

```
1: t := CheckSafe(A,C^may );
2: if t == null then
3:    invoke Oracle 2
4: else
5:    σ := project(t)
6:    result := Query(σ, C)
7:    if result == no then
8:       return σ to L*
9:    else
10:      invoke Oracle 1 (new Preds)
11:   end if
12: end if
```

Algorithm 4. Oracle 2

```
1: t := CheckPermissive(A,C^must )
2: if t == null then
3:    return A as safe & permissive
4: else
5:    σ := project(t)
6:    result := Query(σ, C)
7:    if result == yes then
8:       return σ to L*
9:    else
10:      invoke Oracle 2 (new Preds)
11:   end if
12: end if
```

optimized algorithm directly learns an interface for component C, meaning that answers to queries and conjectures represent component C itself. As a result, the learning process evolves in parallel with the abstraction refinements. Note that the algorithm never actually uses C itself, but rather its finite-state abstractions C^{must} and C^{may}. We use Preds to denote a global set of abstraction predicates.

Queries. The procedure for queries is illustrated by Algorithm 2. At a high level, a query on σ must return no if $\sigma \in L^E(C)$ and yes otherwise. We briefly explain here how we are able to determine to use C^{must} and C^{may} instead of C. From Proposition 1, $L^E(C^{must}) \subseteq L^E(C) \subseteq L^E(C^{may})$. Therefore, if a counterexample is obtained at line 1, it means that $\sigma \in L^E(C^{must})$, which implies that $\sigma \in L^E(C)$, so the query returns no. If no counterexample is obtained at line 4, then it means that $\sigma \notin L^E(C^{may})$, which implies that $\sigma \notin L^E(C)$, so the query returns yes. Otherwise, we know that the counterexample t obtained belongs to C^{may} but not to C^{must} (if it did, then the check on line 1 would not have returned null). t is then used to refine the abstraction.

Conjectures. We use Theorem 1 to answer the conjectures using two oracles, as illustrated in Algorithms 3 and 4.

Oracle 1: Is the conjectured assumption A safe with respect to C^{may}?
Oracle 2: Is A permissive with respect to C^{must}?

Oracle 1 is invoked first. If it finds that A is safe with respect to C^{may}, Oracle 2 gets invoked. If Oracle 2 finds that A is also permissive with respect to C^{must}, we conclude from Theorem 1 that A is a safe and permissive interface for C. All remaining cases require either the refinement of A by L*, or the refinement of the component abstractions. We use queries to help us determine what needs to be refined. Our approach is described in detail below.

Oracle 1: If A is not safe with respect to C^{may}, we obtain a counterexample t, which leads to error in C^{may}. We subsequently query $\sigma = project(t)$ on the component (lines 5 and 6, Algorithm 3); here $project(t)$ denotes the sequence of method calls corresponding to the sequence of transitions in t, so that σ is over the interface alphabet that L* is learning. From line 1, we know that $\sigma \in L(A)$. The querying procedure may involve refinement of the abstraction; let $C^{may'}$ denote the may abstraction used in the last iteration of the query, when it returns. If the query returns no, then it means that σ should not be in the language of A, so σ is returned to L* for A to be refined. Otherwise, we invoke Oracle 1 again, knowing that Preds have been updated. The reason is that, since the result of the query is yes, σ is safe in $C^{may'}$, meaning $\sigma \notin L^E(C^{may'})$ (lines 4 and 5, Algorithm 2), but is unsafe in C^{may}, meaning $\sigma \in L^E(C^{may})$ (line 1, Algorithm 3).

Oracle 2: If A is not permissive with respect to C^{must}, we obtain a counterexample t, which leads to some state (err, s_c) in $A_{\mathsf{err}} \parallel C^{must}_{sink}$, where s_c is a non-error state in C^{must}_{sink}. Therefore t does not lead to error in C^{must}_{sink}. Moreover, $\sigma = project(t)$ is not in $L(A)$. We subsequently query σ on the component (line 6, Algorithm 4). The querying procedure may involve refinement of the abstraction; let $C^{must'}$ denote the must abstraction used in the last iteration of the query, when it returns. If the query returns yes, then it means that σ should be in the language of A, so it is returned to L*. If the query returns no, then we invoke Oracle 2 again, knowing that Preds have been updated. The explanation is as follows. When the query returns no, it means that: 1) σ is unsafe in $C^{must'}$ (line 1, Algorithm 2); and 2) $\sigma \in L^E(C)$. On the other hand, σ must be safe in C^{must}; if σ were unsafe in C^{must}, then the permissiveness check of line 1 could not have returned t as a counterexample, since $\sigma = project(t)$. Therefore clearly, $C^{must'}$ is more refined that C^{must}. Note that since $\sigma \in L^E(C)$ but $\sigma \notin L^E(C^{must})$, t cannot be a trace of C^{must}, but is rather a sink trace in C^{must}_{sink}.

4.2 Abstraction Refinement

In the algorithms BuildInterface and LearnReuse presented above, the abstraction refinement procedure is applied whenever a violating trace t is discovered that belongs to C^{may} but not to C^{must}. Consequently, t must contain a *may* transition $(a_i \xrightarrow{\tau}_{may} a_{i+1})$ that is not a *must* transition. This means that there exists at least another abstract state a'_{i+1} that is a successor of a_i by τ via a *may* transition,

i.e. $a_i \xrightarrow{\tau}_{may} a'_{i+1}$. The reason is that a_i does not distinguish between concrete states of two types: those whose successors are abstracted to a_{i+1} and those whose successors are abstracted to a'_{i+1}.

Automated abstraction refinement consists in adding new abstraction predicates (based on weakest pre-conditions). As a result, we split a_i into two or more new abstract states, corresponding to predicates in $a_i \wedge \mathsf{wp}(a_{i+1}, \tau)$ and $a_i \wedge \neg\mathsf{wp}(a_{i+1}, \tau)$ respectively, that separate the concrete states of type (i) and (ii) above. Note that this results in a finer partition of the concrete states. The new abstraction will no longer contain τ as a *may* and non-*must* transition and therefore we have the following proposition:

Proposition 2. *If trace t has a transition τ that is of type* may *but not* must*, the refined abstraction results in a strictly finer partition and does not contain transition τ.*

In practice, given a sequence of transitions as a counterexample $\mathsf{Cex} = \{\tau_1, \tau_2, \ldots, \tau_n\}$, we compute refinement predicates using wp computations recursively $\mathsf{wp}(true, \mathsf{Cex}) = \mathsf{wp}(\mathsf{wp}(true, \tau_n), \{\tau_1, \tau_2, \ldots, \tau_{n-1}\})$.

Our refinement algorithm uses weakest precondition calculations to compute new abstraction predicates that are guaranteed to eliminate these may transitions, and returns the newly discovered predicates. We note that unlike standard approaches to counterexample-based abstraction refinement [6], we do not refine solely based on "spurious" counterexamples. The counterexamples obtained from failed safety checks may be feasible, but they may still lead to refinement since they contain non-must transitions.

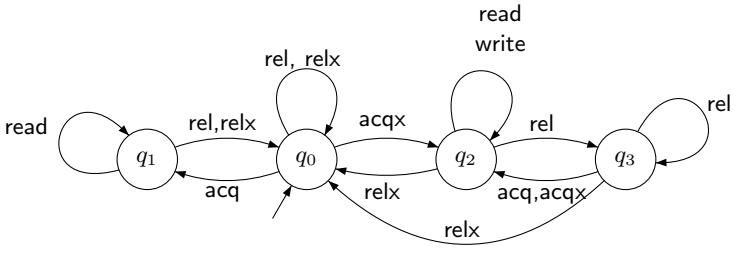


Fig. 2. Read-Write-Acq Example Interface

Example 3. For the example of Figure 1, our algorithms generate the safe and permissive interface A shown in Figure 2. The interface captures the enforcements imposed by the library. Method read can only be called after calling acq (q_1) or acqx (q_2). However, method write can only be called after calling acqx (q_2). Consecutive calls of acq or acqx are inhibited and acq once called can be called again only after calling rel or relx.

The generated interface has one state more than the interface presented in [16] for the same example. On closer inspection, we see that the automaton accepts the string σ = acqx,write,rel,acq,write which is not accepted by their interface. After calling the method acqx from the start state $s_0 = \{x = 0, a = NULL, e = 0\}$,

the variable a becomes non-null and x is set to 1. The method write does not modify a or x. The next method call rel only sets a to $NULL$ but leaves x unchanged (which remains 1). Now after the acq method a is again set to non-null. Since $a \neq NULL$ and $x = 1$, the write method can now be called safely. When we contacted the authors of [16], they observed a discrepancy between the example as it appeared in their paper and their implemented case study, which explains the difference in our respective results.

4.3 Correctness and Termination

In this section we argue the correctness and termination of our algorithms. We will be using *Alg* to represent either BuildInterface or LearnReuse, when our presented arguments hold for both algorithms.

Theorem 2 (Correctness). *If algorithm Alg terminates (with final abstractions C^{must} and C^{may}), then the constructed interface A is* safe *and* permissive *for C. Furthermore, $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$.*

Termination. For infinite-state components, the predicate abstraction refinement used in *Alg* may not always terminate. However, we can make the following termination argument:

Theorem 3. *If Alg computes an abstraction such that $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$, then the Alg* terminates.

Furthermore, from previous work on automatic abstraction refinement [22,19], we know that if the component C has a finite bisimulation quotient [20], then the refinement based on weakest precondition calculations is guaranteed to converge to that finite quotient.

Theorem 4 (Bisimulation Completeness [22,19]). *If the component C has a finite bisimulation quotient, then there exists a refinement iteration bound such that the abstraction C^{may} is bisimilar to C.*

Since bisimulation implies trace equivalence [21] and from Proposition 1, it follows that if C has a finite bisimulation quotient, then there exists a refinement iteration bound such that for the obtained set of abstraction predicates, $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$. Therefore, together with Theorem 4 we conclude the following:

Theorem 5 (Termination). *If the component C has a finite bisimulation quotient then Alg* terminates.

We observe that this termination condition is not very tight as our algorithms also terminate for systems for which predicate abstraction with refinement results in an abstraction such as $L^E(C^{must}) = L^E(C^{may}) = L^E(C)$, which is a weaker condition than the existence of a finite bisimulation quotient (see Theorem 3).

Let us finally note that although in general our algorithms may not terminate, they can be made to return results "any time". At any stage, we may use L* to compute interfaces for A^{may} for C^{may} and A^{must} for C^{must}. The language of

the safe and most permissive interface A for component C is bounded between the languages of A^{may} and A^{must}.

5 Implementation and Experiments

Implementation. We have implemented the algorithms presented in Section 4 in the ARMC tool [24]. ARMC already had support for may abstractions; we extended it with support for must abstractions. Furthermore, ARMC provides abstraction refinement algorithms based on Craig interpolation [17]. We have integrated these algorithms in our approach, as an alternative to refinement based on weakest preconditions.

We note that the algorithms presented previously use the explicit composition of the abstraction with the interface. Instead of performing this explicit composition, our implementation builds the abstract graph of the composite automaton implicitly, by method inlining. This helps us avoid un-necessary computation and only constructs a part of component abstractions which are sufficient to prove (or disprove) the *safety* and *permissive* checks.

We observe that in the basic algorithm, only *feasible* counterexample traces can add error behaviors to the must abstraction C^{must}. The spurious counterexamples only remove error behaviors from the may abstraction C^{may}. Therefore it suffices to restart learning only after refining feasible counterexamples. In the case of spurious counterexamples, the CheckSafe algorithm is restarted after the may abstraction is refined; it terminates when either a feasible counterexample is found or the interface is discovered to be safe.

Experiments. We evaluate our interface generation algorithm on several sample Java2SDK library classes presented in [1,17] as well as some benchmarks from J2SEE and the NASA CEV 1.5 EOR-LOR mission profile case study [12]. A brief description about the modelling and generated interfaces follows. The experiments were run on a dual core 1.80 GHz Intel Pentium processor with 3 GB of RAM. Table 1 presents the empirical results obtained from following different algorithmic schemes: **wp:** BuildInterface with weakest precondition refinement; **wp+craig:** BuildInterface with craig interpolation refinement for infeasible counterexamples (wp+craig); **refine-may:** BuildInterface with refining only may abstraction for infeasible counterexamples (refine-may + craig); **learn-reuse:** LearnReuse with craig. The table also presents the number of predicates (#Preds) discovered, the number of learning iterations (#Iterations), the number of states in the final interface (#States) and the running times.

The primary purpose of our experiments is to assess the feasibility of our approach. We additionally provide, with a smaller emphasis, a comparison between algorithms BuildInterface and LearnReuse. Our results show that the proposed approach is feasible, and also quantify the expected improvement achieved by LearnReuse. We can additionally use our experimental results as approximate indications of the practical savings achieved by LearnReuse over previous approaches that perform learning and abstraction separately [1]. These approaches

are based on manual refinement, but if automated, their performance would be similar to BuildInterface since they do not perform abstraction on demand during the learning process.

For the Signature class we selected five methods as the alphabet Σ = {initSign, initVerify, sign, update, verify}. The exception SignatureException was modelled as the error predicate. The states in the generated interface correspond to the labellings of uninitialized, sign and verify respectively. The ServerTableEntry class is taken from the package com.sun.corba.se.internal.Activation. We selected six methods as the alphabet Σ ={activate, register, registerPorts, install, uninstall, holdDown} and modelled the exception INTERNAL as the error condition. The generated interface only keeps track of three states: activated(register), running(install/uninstall) and other states as one state. The ListItr class is an inner class of AbstractList from the package java.util. We selected five methods as the alphabet Σ ={next, remove, previous, set, add} and the exception IllegalStateException was modelled as the error predicate. The interface captures the inhibition of calls of methods set and remove after calling methods remove or add.

The PipedOutputStream class is taken from the package java.io and is an implementation of an abstract class OutputStream. We selected five methods as the alphabet Σ ={close, (connect,0), (connect,1), flush, write }, where we model invocations of connect method returning different values (0 or 1) as different methods ((connect,0) or (connect,1)) similar to the approach taken in [5]. The exception NullPointerException was modelled as the error predicate. The interface captures precisely two states where sink = null and sink $\neq$ null. Only a successful connect call can enable flush and write methods. The Socket class is part of java.net package which implements client sockets. We considered seven methods as the alphabet Σ = {close, bind, getInputStream, getOutputStream, shutdownInput, shutdownOutput } and the exception SocketException was modelled as the error predicate. The interface enforces the requirement that bind cannot be called after connect, shutdownInput can only be called after calling connect, getInputStream can only be called after the connect call and not after close or shutdownInput has been called. After calling close no other method calls are allowed. The class TransactionManager is taken from the package javax.transaction, and we selected six methods as the alphabet Σ = {begin, suspend, resume, commit, rollback, setrollbackonly}. The exception IllegalStateException was modelled as the error predicate. The generated interface captures the precise sequence of method calls for performing a transaction with appropriate handling of commit and rollback actions.

We also applied our technique to obtain the interface for the simplified state machine for NASA CEV 1.5 EOR-LOR mission profile case study. It models the Ascent,EarthOrbit,TransitEarthMoon and Entry phases of the space-craft, the events (like srbIgnition,stage1separation etc.), the vehicle configuration and various failure modes. The Java model is avaiable with the JPF distribution under examples/jpfESAS. We modelled the 22 events as the alphabet set for the interface and the error predicate was modelled as the failure modes and the event calls from inappropriate states. Events with parameters like abort(boolean

Table 1. Experiment results on benchmark case studies

Class name	Algorithm	#Preds	#Iterations	#States	Running Time
ListItr	wp	12	5	2	40.6s
	wp+craig	**7**	6	2	42.2s
	refine-may	8	**4**	2	39.3s
	learn-reuse	6	1	2	**12.7**
Signature	wp	8	9	3	72.7s
	wp+craig	**5**	6	3	42.9s
	refine-may	7	**4**	3	33.2s
	learn-reuse	5	1	3	**16.6s**
ServerTableEntry	wp	10	10	3	98.1s
	wp+craig	**6**	7	3	64.9s
	refine-may	10	**5**	3	51.3s
	learn-reuse	7	1	3	**19.2**
PipedOutputStream	wp	4	5	2	16.4s
	wp+craig	2	3	2	11.4s
	refine-may	2	3	2	11.6s
	learn-reuse	2	1	2	**7.4s**
read-write-acq	wp	6	6	4	122.8s
	wp+craig	**4**	5	4	75.4s
	refine-may	7	5	4	74.3s
	learn-reuse	6	1	4	**31.1**
Socket	wp	25	5	6	468.5s
	wp+craig	13	5	6	272.9s
	refine-may	13	5	6	228.0s
	learn-reuse	**12**	1	6	**65.8**
TransactionManager	wp	15	8	4	138.6s
	wp+craig	9	7	4	103.5s
	refine-may	9	4	4	76.9s
	learn-reuse	9	1	4	**30.4s**
NASA-Ascent	wp	34	14	5	1685.6s
	wp+craig	20	14	5	1433.9s
	refine-may	20	6	5	712.4s
	learn-reuse	20	1	5	**75.6s**
NASA-Complete	learn-reuse	74	1	14	3115.6s

controlMotorField) were modelled as two events one with controlMotorField parameter true (abortctr) and the other with controlMotorField parameter false (abortnctr) which increased the alphabet size to 26. The interface has 14 states and required 74 predicates. For such large interfaces, only the LearnReuse algorithm finished in reasonable time. Table 1 also documents the results for NASA-Ascent interface where only the Ascent phase of the space-craft was modelled. These interfaces in addition to being helpful in verifying the space-craft code are also a useful tool to help debug the system early in the designing process of such critical software.

References

1. Alur, R., Cerný, P., Madhusudan, P., Nam, W.: Synthesis of interface specifications for java classes. In: POPL, pp. 98–109 (2005)
2. Alur, R., Henzinger, T.A., Mang, F.Y.C., Qadeer, S., Rajamani, S.K., Tasiran, S.: Mocha: Modularity in model checking. In: Y. Vardi, M. (ed.) CAV 1998. LNCS, vol. 1427, pp. 521–525. Springer, Heidelberg (1998)
3. Ammons, G., Bodík, R., Larus, J.R.: Mining specifications. In: POPL, pp. 4–16 (2002)
4. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
5. Beyer, D., Henzinger, T.A., Singh, V.: Algorithms for interface synthesis. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 4–19. Springer, Heidelberg (2007)
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. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL, pp. 238–252. ACM, New York (1977)
8. Das, M., Lerner, S., Seigle, M.: Esp: Path-sensitive program verification in polynomial time. In: PLDI, pp. 57–68 (2002)
9. de Alfaro, L., Godefroid, P., Jagadeesan, R.: Three-valued abstractions of games: Uncertainty, but with precision. In: LICS (2004)
10. Flanagan, C., Freund, S.N., Qadeer, S.: Thread-modular verification for shared-memory programs. In: Le Métayer, D. (ed.) ESOP 2002. LNCS, vol. 2305, pp. 262–277. Springer, Heidelberg (2002)
11. Foster, J.S., Terauchi, T., Aiken, A.: Flow-sensitive type qualifiers. In: PLDI, pp. 1–12 (2002)
12. Giannakopoulou, D., Pasareanu, C.S.: Interface generation and compositional verification in javapathfinder. In: Chechik, M., Wirsing, M. (eds.) FASE 2009. LNCS, vol. 5503, pp. 94–108. Springer, Heidelberg (2009)
13. Godefroid, P., Huth, M., Jagadeesan, R.: Abstraction-based model checking using modal transition systems. In: Larsen, K.G., Nielsen, M. (eds.) CONCUR 2001. LNCS, vol. 2154, p. 426. Springer, Heidelberg (2001)
14. Godefroid, P., Huth, M., Jagadeesan, R.: A game-based framework for ctl counterexamples and 3-valued abstraction-refinement. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 275–287. Springer, Heidelberg (2003)
15. Godefroid, P., Nori, A.V., Rajamani, S.K., Tetali, S.D.: Compositional may-must program analysis: Unleashing the power of alternation. In: POPL (2010)
16. Henzinger, T.A., Jhala, R., Majumdar, R.: Permissive interfaces. In: ESEC/SIGSOFT FSE, pp. 31–40 (2005)
17. Henzinger, T.A., Jhala, R., Majumdar, R., McMillan, K.L.: Abstractions from proofs. In: POPL, pp. 232–244 (2004)
18. Jones, C.B.: Specification and design of (parallel) programs. In: IFIP Congress, pp. 321–332 (1983)
19. Lee, D., Yannakakis, M.: Online minimization of transition systems. In: ACM Symposium on Theory of Computing (1992)
20. Lee, D., Yannakakis, M.: Online minimization of transition systems (extended abstract). In: STOC, pp. 264–274. ACM, New York (1992)

21. Milner, R.: Communication and Concurrency. Prentice Hall, New York (1989)
22. Namjoshi, K.S., Kurshan, R.P.: Syntactic program transformations for automatic abstraction. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 435–449. Springer, Heidelberg (2000)
23. Pnueli, A.: In transition from global to modular temporal reasoning about programs. In: Logics and models of concurrent systems, pp. 123–144 (1985)
24. 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 (2006)
25. Rivest, R.L., Schapire, R.E.: Inference of finite automata using homing sequences. Inf. Comput. 103(2), 299–347 (1993)
26. Shoham, S., Grumberg, O.: Compositional verification and 3-valued abstractions join forces. In: Riis Nielson, H., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 69–86. Springer, Heidelberg (2007)
27. Tkachuk, O., Dwyer, M.B.: Adapting side effects analysis for modular program model checking. In: ESEC / SIGSOFT FSE, pp. 188–197 (2003)
28. Whaley, J., Martin, M.C., Lam, M.S.: Automatic extraction of object-oriented component interfaces. In: ISSTA, pp. 218–228 (2002)

A Dash of Fairness for Compositional Reasoning*

Ariel Cohen[1], Kedar S. Namjoshi[2], and Yaniv Sa'ar[3]

[1] New York University, New York, NY
arielc@cs.nyu.edu
[2] Bell Labs, Alcatel-Lucent, Murray Hill, NJ
kedar@research.bell-labs.com
[3] Weizmann Institute of Science, Rehovot, Israel
yaniv.saar@weizmann.ac.il

Abstract. Proofs of progress properties often require fairness assumptions. Directly incorporating global fairness assumptions in a compositional method is difficult, given the local flavor of such reasoning. We present a fully automated local reasoning algorithm which handles fairness assumptions through a process of iterative refinement. Refinement strengthens local proofs by the addition of auxiliary shared variables which expose internal process state; it is needed as local reasoning is inherently incomplete. Experiments demonstrate that the new algorithm shows significant improvement over standard model checking.

1 Introduction

Model checking is fundamentally constrained by state explosion [6]: for concurrent programs, the state space can grow exponentially with the number of processes. A promising approach to ameliorating state explosion is to decompose a verification task so that the reasoning is as localized as possible. In this work, we propose and evaluate a new algorithm which carries out compositional reasoning for temporal properties which hold only under global fairness assumptions.

Fairness assumptions are often needed for proofs of progress properties. It has long been understood how to incorporate fairness in standard model checking [5,14], but doing so is a challenge for compositional methods. The difficulty is that fairness assumptions commonly refer to local state from a number of processes. For example, a common (strong) fairness constraint is that "*for every process: if the process is enabled infinitely often, it is infinitely often executed*". As "enabledness" depends on local state, this assumption refers to the local state of every process. Since compositional reasoning is based on a per-process view, the presence of such global assumptions can be problematic.

This work develops a new algorithm for compositional model checking with fairness assumptions, which tackles this problem with a successive refinement method. It also

* This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant awarded to David Harel from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 543–557, 2010.

presents a new compositional proof rule for verification under fairness. Moreover, the model checking algorithm can be instrumented to generate a valid instantiation of the proof rule upon success.

The new algorithm is the continuation of a line of research on mechanizing assertional (i.e., state-predicate based) compositional verification. The starting point is an algorithm from [22] which computes the strongest *split invariant*. A split invariant is a vector of interference-free, per-process invariants. (A set of per-process invariants is free of interference [24,19] if the action of one process does not invalidate the invariant of another process.) The term "split invariant" is used as the conjunction of the local invariants forms a inductive invariant for the program as a whole. The strongest split invariant may be weaker than the set of reachable states, and therefore not strong enough to prove a safety property. In [8], we solve this problem by formulating an complete verification procedure which strengthens the split invariant by discovering and adding auxiliary shared variables to track local predicates. In [9], we use split invariance as the basis for a new compositional algorithm for checking LTL properties. Experiments reported in these papers show that assertional local reasoning can be significantly faster than monolithic (i.e., non-compositional) model checking.

The local liveness method of [9] does not directly apply to fairness constraints. This is because the method is sound only for properties expressed over shared variables. Incorporating fairness into the specification, through the identity $M \models \mathsf{A}_{Fair}(Spec) \equiv M \models \mathsf{A}(Fair \Rightarrow Spec)$, results in a new specification which names a number of local variables (due to $Fair$). One can, of course, turn all the local variables in $Fair$ into shared variables, but this defeats the purpose of local reasoning.

The new algorithm gets around this difficulty by a process of iterative refinement. The fairness constraint is replaced with a weaker form, which depends monotonically on the current split invariant, and is expressed over only the shared variables. This allows using the compositional algorithm from [9], with slight modifications. If verification succeeds with the weaker fairness assumption, the property is proved. If not, a bogus counter-example is produced, and analyzed to discover new local predicates which are then exposed as auxiliary shared variables. Exposing local state strengthens the split invariant in the next round of computation, which strengthens the abstracted fairness assumption by monotonicity. This is repeated until a decisive result (either success or a real counter-example) is obtained. The iterative process terminates—and is thus complete—for finite-state programs: eventually, enough of the local state is exposed to either prove a correct property or to disprove an incorrect one. Moreover, it is possible to disprove a property *without* building up the entire state space.

The algorithm, being predicate-based, has a simple implementation using BDDs. We carry out an evaluation with several parameterized protocols, where each instance of the protocol is finite-state. The experimental results show promise: the compositional verification is faster in almost all cases, sometimes by one or two orders of magnitude. Exposing a limited amount of local state suffices for both proofs and disproofs of properties, validating the basic premise behind compositional reasoning.

2 Related Work

The question of handling fairness in compositional verification is a natural and important one. The comprehensive book on compositional methods by de Roever et. al. [11], however, does not mention a compositional proof rule directly incorporating fairness. Compositional proof rules for general LTL properties (e.g., [1,20,21,23]) can handle fairness only by compiling it into the specification. To the best of our knowledge, this is the first compositional algorithm and proof rule to directly incorporate fairness.

The methods used here are assertional; i.e., they are based on computing state predicates. The "thread-modular" reasoning method [16] computes a split invariant using explicit-state representations, but is limited to safety properties. An alternative line of work on automated compositional reasoning is based on representing interface behavior, and is thus behavioral in nature. One instance of this method uses the following complete proof rule: to show $M_1//M_2 \models Spec$, find an interface automaton A such that $M_1 \models A$ and $M_2//A \models Spec$. (Here, $\models$ is read as language inclusion.) The procedures developed in [17,27] employ a combination of model checking and finite-automaton learning via variants of the L^* method [2] to construct an appropriate automaton A. Standard learning algorithms compute automata on finite words, and hence can be used only for proofs of safety properties. An algorithm is developed in [15] for learning a Büchi automaton, but it has not yet been applied to verification of progress properties. Although an automaton is a powerful and compact representation object, current implementations of behavioral methods have difficulty showing a significant improvement over non-compositional model checking [7].

3 Background: Local Reasoning and Liveness Properties

This section defines the system model and split invariance, and gives a short summary of the method for local liveness checking. Part of this material is taken from [22,8,9], and is repeated here for convenience.

A Note on Notation. Throughout the paper, we use notation based on that of Dijkstra and Scholten [12]. Sets of program states are represented by first-order formula on program variables. Existential quantification of formula ξ by a set of variables X is written as $(\exists X : \xi)$. The notation $[\xi]$ stands for "ξ is valid". The successor operation is denoted by sp (for strongest post-condition): $sp(\tau, \xi)$ represents the set of states reachable from states satisfying ξ in one τ-transition. The notation $sp_i(\tau, \xi)$ is used for successors computed within the state space of process P_i.

3.1 Model: Asynchronous, Shared-Memory Composition

A *process* is given by a tuple (V, I, T), where V is a set of (typed) variables, $I(V)$ is a predicate over V defining an initial condition, and $T(V, V')$ is a predicate defining a transition condition, where V' is a fresh set of variables in 1-1 correspondence with V. The semantics of a process is given by a *transition system* in the standard way.

The *asynchronous composition* of processes $\{P_i\}$ is written as $\|_i P_i$. For convenience, we suppose that there is a set of variables, X, called the *shared variables*, and

sets of variables, $\{L_i\}$, called the *local variables*, such that $V_i = X \cup L_i$ for each i, and L_i is disjoint from L_j, for $i \neq j$, and L_i is also disjoint from X. The components of the composition are defined as follows. Let $V = \bigcup_i V_i$ and $I = \bigwedge_i I_i$. The set of local variables is $L = \bigcup_i L_i$. Let $\hat{T}_i = T_i(V_i, V_i') \wedge (\forall j : j \neq i \;\Rightarrow\; unch(L_j))$, where $unch(W)$ is short for $\bigwedge_{w \in W}(w' = w)$. Thus, $\hat{T}_i$ behaves like T_i, but leaves local variables of other processes unchanged. The transition relation of the composition, T, is defined as $\bigvee_i \hat{T}_i$.

3.2 Split-Invariance: Definition and Calculation

Let $P = ||_k P_k$ be an N-process composition. For localized reasoning about invariance, the shape of invariance assertions is restricted to a conjunction of local (i.e., per-process) assertions. A *local* assertion is one that is based on the variables of a single process. A *split assertion* is a vector of local assertions, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$, one for each process, so that θ_i is defined on V_i (equivalently, on X and L_i). Split assertion θ is a *split invariant* if the conjunction of its components, i.e., $\bigwedge_k \theta_k$, is an inductive invariant for the full program P. Split-invariance can equivalently be defined as in Figure 1.

Definition 1. *The notation $T_k^\theta(X, X')$ denotes $(\exists L_k, L_k' : T_k \wedge \theta_k)$. This is a "summary transition", representing the effect that a move of P_k from a state satisfying its local invariant has on the shared variables.*

For each process index i:

1. **[initiality]** θ_i includes all initial states of process P_i. I.e., $[(\exists L \setminus L_i : I) \;\Rightarrow\; \theta_i]$
2. **[step]** θ_i is closed under transitions of P_i. I.e., $[sp_i(T_i, \theta_i) \;\Rightarrow\; \theta_i]$
3. **[non-interference]** θ_i is closed under transitions (interference) by processes other than P_i. I.e., for all k different from i, $[sp_i(T_k^\theta \wedge unch(L_i), \theta_i) \;\Rightarrow\; \theta_i]$

Fig. 1. Split Invariance Conditions

These conditions are a simple instance of (syntactically circular) *assume-guarantee reasoning*: θ_i is the invariance guarantee provided by process i, based on assumptions $\{\theta_j : j \neq i\}$ about the other processes. The constraints can be gathered into the set of simultaneous implications: for each i,

$$[(\exists L \setminus L_i : I) \vee sp_i(T_i, \theta_i) \vee (\vee k : k \neq i : sp_i(T_k^\theta \wedge unch(L_i), \theta_i)) \;\Rightarrow\; \theta_i] \quad (1)$$

Theorem 1. *(Namjoshi [22]) The simultaneous least fixpoint of equations (1) exists by the Knaster-Tarski fixpoint theorem. This defines the strongest split invariant.*

3.3 Incompleteness and Auxiliary Variables

Local reasoning is inherently incomplete. This is illustrated by the mutual exclusion protocol from Figure 2. The strongest split invariant for 2 processes is $(true, true)$, which is too weak to prove mutual exclusion. A general mechanism for overcoming

x : **boolean initially** $x = 1$

$$\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0 : \textit{Non-Critical} \\ l_1 : \textbf{request}\ x \\ l_2 : \textit{Critical} \\ l_3 : \textbf{release}\ x \end{array}\right] \end{array}\right]$$

Fig. 2. MUXSEM

last : **0..N initially** last $= 0$

x : **boolean initially** $x = 1$

$$\|_{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0 : \textit{Non-Critical} \\ l_1 : \langle \textbf{request}\ x;\ \text{last} := i \rangle \\ l_2 : \textit{Critical} \\ l_3 : \textbf{release}\ x \end{array}\right] \end{array}\right]$$

Fig. 3. MUXSEM with auxiliary variable

Local Liveness (LL) Algorithm

1. Compute the strongest split invariant, θ.
2. For each i: build an abstract form of process i, called P_i^θ, with initial states given by $(\exists L \setminus L_i : I)$, and two kinds of transitions:
   - the transition T_i of process i, and
   - summary transitions T_j^θ (see Defn. 1) for all other processes P_j $(j \neq i)$
3. Form a Büchi automaton for the *negated* specification. For each i, form the synchronous product of this automaton with P_i^θ and check that *there is no computation where infinitely often there is a process i transition from a Büchi accepting state*
4. Declare success if the check succeeds **for each** abstract process

Fig. 4. Local Liveness (LL) Algorithm

incompleteness, proposed by Owicki-Gries and Lamport [19], is to add auxiliary shared variables which expose portions of the local state or execution history. In Figure 3, an auxiliary variable records the last process to enter the critical section. The strongest split invariant for the augmented protocol is given by $\theta_i \equiv (l_2(i) \equiv (x = 0) \wedge (last = i))$, which suffices to prove mutual exclusion as $[\theta_i \wedge \theta_j \wedge (i \neq j) \Rightarrow \neg(l_2(i) \wedge l_2(j))]$. The discovery of auxiliary predicates can be effectively automated [8].

3.4 Local Verification of Liveness Properties

Owicki and Gries also developed compositional proof rules for termination. In [9], a related proof rule is turned into a compositional algorithm for checking general linear-time temporal properties. This "local liveness" method, referred to subsequently as the LL algorithm, is shown in Figure 4. We give a sketch of its soundness proof, as this is important for the extension to fairness. The LL algorithm requires that the LTL property is expressed by shared variables. With this method, one can show that the property "infinitely often $(x = 0)$" holds for the protocol in Figure 2—i.e., that some process is in the critical section infinitely often. Starvation freedom, however, holds only under a strong fairness assumption, and its compositional proof requires the new method.

Theorem 2. *(Cohen-Namjoshi [9]) The LL method is sound.*

Proof Sketch. The soundness proof shows the following: if a property does not hold, any global counter-example can be projected to a counter-example for **some** abstract process. Let σ be a global counter-example. Then (1) each state of σ must satisfy the

split invariant and (2) the Büchi automaton must accept infinitely often along σ. As there is a fixed number of processes, by (2), there is a process, say P_i, whose transition is executed infinitely often from a Büchi accepting state along σ. Consider the abstract process P_i^θ formed out of P_i. The computation σ can be projected, transition-by-transition, to an execution of P_i^θ. A transition by process P_i is kept as is; a transition by another process, say P_k, is replaced by its summary transition, T_k^θ (detailed proof is in [9]). Any summary transition preserves the change to shared variables made by the original; hence, the sequence of shared-variable values is identical in the original and the projected computations. As the automaton checks properties defined only over shared variables, its accepting run carries over to the projected computation. In the projected computation, there are infinitely many positions where there is a transition by P_i from an accepting automaton state. Hence, the check in Step 3 fails for process P_i^θ. □

4 Fairness

We describe the modifications necessary to incorporate fairness assumptions into the local liveness method. We begin with a simple but useful kind of fairness, called *unconditional fairness*.

4.1 Unconditional Fairness

This fairness notion is a foundational concept in the UNITY programming language and proof system [3], and it suffices for many interesting distributed protocols. Under unconditional fairness, every process is scheduled infinitely often in an infinite computation. The statement uses "scheduled" rather than "executed"—a process may be scheduled but do nothing (i.e., behave as *skip*) because its transition is not enabled. To analyze a protocol under unconditional fairness, Step 3 of the local liveness method is modified to check that, for each P_i^θ, *there is no unconditionally fair computation where infinitely often there is a process i transition from a Büchi accepting state*.

Theorem 3. *The* LL *method modified for unconditional fairness is sound.*

Proof Sketch. The proof sketch for Theorem 2 shows that the sequence of process identifiers associated with the transitions is identical in the original and the projected computations. As the original error computation is unconditionally fair by assumption, the projected error computation must also be unconditionally fair. This argument shows that the modified check is sound. □

4.2 Strong Fairness

The strong fairness algorithm is based on iterated refinement. The idea is to start with a weakened form of the strong fairness assumption, and use the refinement mechanism which adds auxiliary variables to strengthen this assumption with each iteration, until a conclusive result is obtained. To keep the notation simple, we consider a common form of strong fairness, given as $\Phi \equiv (\bigwedge i : \mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$, where p_i and q_i are assertions over the variables of process P_i. Recall that the proof of soundness of the

local liveness method projects a global counter-example, σ, on to a local computation of abstract process P_k^θ, for some k. In the presence of fairness, there are two key properties of σ:

1. Every state on σ satisfies $\bigwedge_i \theta_i$, as θ is a split invariant, and
2. σ satisfies the fairness assumption Φ

Taken together, this implies—crucially—that σ must also satisfy the *stronger* fairness assertion, Φ^*, given by $(\bigwedge i : \mathsf{FG}(\theta_i \wedge p_i) \vee \mathsf{GF}(\theta_i \wedge q_i))$. The fact that Φ^* is stronger than Φ for any θ follows from the monotonicity of G and F. The fact that Φ^* holds for σ follows by the first property: as every state on σ satisfies $(\bigwedge j : \theta_j)$, assertions $\mathsf{FG}(\theta_i \wedge p_i)$ and $\mathsf{FG}(p_i)$ are equivalent on σ, as are assertions $\mathsf{GF}(\theta_i \wedge q_i)$ and $\mathsf{GF}(q_i)$.

The abstract fairness property is formed by quantifying out local variables from Φ^*, as follows.

$$\Phi^\theta = (\bigwedge i : \mathsf{FG}((\exists L_i : \theta_i \wedge p_i)) \vee \mathsf{GF}((\exists L_i : \theta_i \wedge q_i)))$$

Subsequently, we refer to the term $(\exists L_i : \theta_i \wedge p_i)$ as p_i^θ and to $(\exists L_i : \theta_i \wedge q_i)$ as q_i^θ. The transformed fairness property is weaker than Φ^*, but not necessarily weaker than Φ, and it is defined over the shared variables only.

It is important that Φ^θ depends on θ, and does so in a monotonic manner. This enables refinement: as the split invariant is strengthened by adding auxiliary variables, the abstract fairness assumption also becomes stronger. The new method is shown in Figure 5; other than a modified check at Step 3, it is identical to the LL method from Figure 4.

Theorem 4. *The FLL method is sound.*

Proof. This proof is an extension of the proof of Theorem 2. Consider a global counter-example σ which is fair according to Φ. By the proof of Theorem 2, the projection of σ on P_i^θ satisfies the second part of the condition of Step 3: i.e., infinitely often there is a process i transition from a Büchi accepting state. It remains to be shown that the projected computation also satisfies Φ^θ.

As σ is a counter-example based on the fairness assumption, it satisfies Φ; as it is a program computation, it satisfies the split invariant, θ. Hence, by the reasoning above, it satisfies Φ^* and therefore the weaker property Φ^θ. As Φ^θ is a property over shared state only, and the sequence of values for shared variables is preserved by the projection, Φ^θ holds also of the projected computation. □

Fair Local Liveness (FLL) Algorithm

1. Compute the strongest split invariant, θ.
2. For each i: build an abstract form of process i, P_i^θ, as defined in Figure 4.
3. Form a Büchi automaton for the *negation* of the specification. For each i, form the synchronous product of this automaton with P_i^θ and check that *there is no computation which is strongly fair according to Φ^θ and on which infinitely often there is a process i transition from a Büchi accepting state*
4. Declare success if the check succeeds **for each** abstract process

Fig. 5. Fair Local Liveness (FLL) Algorithm

Refinement for Fair Local Liveness

1. Check if every summary transition in the abstract counter-example σ is a MUST transition for the process which makes it. If not, expose a local predicate for the MUST condition, as defined in [9] for the LL method, and REPEAT the full verification.
2. Inductively construct a global computation δ which matches σ
3. Check if δ satisfies the original fairness condition, Φ. If so, HALT with δ as the valid global counter-example.
4. Use a fairness term $(\mathsf{FG}(p_j) \vee \mathsf{GF}(q_j))$ which is *not* satisfied by δ to discover and expose a local predicate, and REPEAT full verification.

Fig. 6. Refinement for the FLL method, given a counter-example σ in the abstract process P_i^θ

Remark 1. Our implementation uses a stronger abstraction of the fairness property. In Step 3 of the FLL algorithm, instead of the uniform assumption Φ^θ, the implementation uses a fairness assumption for P_i^θ where all terms from Φ are abstracted relative to θ as described above, *except* the term $(\mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$, which is used as is, since it refers only to variables of process P_i.

4.3 FLL Algorithm Variant

The basic FLL algorithm can be varied by changing Steps (2)-(4) as follows. The new combination checks whether for *some* i, *the abstract process* P_i^θ *satisfies the specification, assuming strong fairness according to* Φ^θ. We call this algorithm the B-variant of the FLL algorithm; the original is called the A-variant. Note that the correctness condition in FLL (B) is stricter than that for FLL (A); on the other hand, it suffices that one of the abstract processes satisfies the test. The justification is based on a proof similar to that of Theorem 4: if a global counter-example exists, its projection in P_i^θ fails the FLL (B) requirement, for *every* i. The contra-positive shows that it suffices for some i to satisfy the FLL (B) requirement for the program to be correct.

The two algorithms offer a trade-off. Due to the weaker correctness condition of FLL (A), this algorithm may prove correctness while FLL (B) does not, leading to extra refinements in the B-variant. On the other hand, for FLL (B), it suffices to check a single, fixed process (say, P_0^θ); this is potentially faster for programs with a large number of components.

4.4 Refinement for Fairness

As local reasoning is approximate, it is possible for the FLL method to fail even though the property is true of the whole program. One can analyze the failure, though, to suggest auxiliary Boolean variables which expose local state predicates, as shown in Figure 6, which extends the refinement procedure used for the LL method.

Step 1 is the refinement step for LL. Recall that a transition of σ in P_i^θ by a process P_k other than P_i can modify only the shared variables. A change of shared state from $X = a$ to $X' = b$ is considered a MUST transition if this change is possible no matter what the local state of process P_k may be, so long as it is consistent with θ_k. The

predicate $m(L_k) \equiv \theta_k(a, L_k) \wedge \neg(\exists L'_k : T_k(a, L_k, b, L'_k))$ expresses this succinctly: the transition from $X = a$ to $X' = b$ is a MUST transition if, and only if, m is unsatisfiable. If m is satisfiable, it is "exposed" by adding an auxiliary shared variable x_m. The constraint $x'_m \equiv m(L'_k)$ is added to the transition relation of P_k, and the constraint $x'_m \equiv x_m$ to that for all other processes. Together with the initialization of x_m to $m(L_k)$, these constraints maintain the global invariant $(x_m \equiv m)$.

Regarding Step 2, if each summary transition in σ is a MUST transition, it is possible to inductively construct a global computation δ which matches σ. The initial values for the local variables for processes other than P_i can be chosen arbitrarily, consistent with the initial condition. Inductively, the MUST property guarantees that a concrete transition can be found for each process making a summary move such that the change to the shared state is preserved. Although σ satisfies Φ^θ, it need not be the case that δ satisfies Φ. If Φ fails to hold on the computed δ (Step 3), the proof of Theorem 6 shows how a new predicate can be derived by analyzing this failure.

Theorem 5. *(Soundness for failures) If the FLL refinement procedure halts with failure, the trace is a valid counter-example under strong fairness.*

Proof. Follows from the reasoning given for Steps 2 and 3. □

Theorem 6. *(Finitary Completeness) The FLL procedure with refinement terminates for finite-state programs.*

Proof. It suffices to show that a new predicate—one that is not a Boolean combination of existing predicates—is added at each refinement step. Termination follows, as there is a finite number of distinct predicates. Theorems 4 and 5 show that each termination outcome is correct; thus, the method is complete. For Step 1, the fact that a new predicate is added was shown for the LL method in [9]. For the predicate added at Step 4, it can be shown as follows.

If the check at Step 3 fails, there is a term, $(\mathsf{FG}(p_j) \vee \mathsf{GF}(q_j))$, for some j, which fails to hold for δ. Thus, from some point on, all states on δ fail q_j, and infinitely often, there is a state failing p_j. Depending on which sub-term is used to satisfy $(\mathsf{FG}(p_j^\theta) \vee \mathsf{GF}(q_j^\theta))$ on σ, there is a state s that is on σ and its corresponding state t on δ such that either (i) s satisfies p_j^θ and t does not satisfy p_j or (ii) s satisfies q_j^θ while t does not satisfy q_j.

Consider the first case, the proof of the second is similar. By the definition of p_j^θ as $(\exists L_j : \theta_j \wedge p_j)$, there is a valuation c for L_j such that for $u = (s(X), c)$ it is the case that $\theta_j(u)$ and $p_j(u)$ both hold. On the other hand, while $\theta_j(t)$ holds by the invariance of θ for the concrete computation δ, $p_j(t)$ does not hold by the assumption. By the correspondence of s and t, states u and t differ only on the valuation of L_j. Let q be a predicate expressing this difference (e.g., $q(L_j) = (L_j = c)$). We have to show that q is a new predicate; i.e., it cannot be expressed as a function of the already exposed predicates.

A property of the split invariant, which can be shown by induction, is that $[\theta_j \Rightarrow (x_m \equiv m)]$ for each shared refinement variable x_m that is added for a predicate m exposed for process P_j. As u and t agree on all shared variables, including refinement variables, and as both satisfy θ_j, it follows that all prior predicates exposed for P_j have identical values on u and t. As this is not true for q, it cannot be expressed as a function of the already exposed predicates. □

4.5 Weak and Generalized Fairness

Weak Fairness, also called "justice", has the normal form $\mathsf{GF}(p)$ ("infinitely often p"). It is often used to express the constraint that a continuously enabled transition cannot be forever ignored; i.e., $\mathsf{FG}(enabled) \Rightarrow \mathsf{GF}(executed)$. As its normal form is a special case of strong fairness, the algorithm developed for strong fairness can be applied to it. Thus, the common weak fairness specification $\Phi \equiv (\bigwedge i : \mathsf{GF}(p_i))$, where p_i is an assertion over the variables of process P_i, is abstracted to $\Phi^\theta \equiv (\bigwedge i : \mathsf{GF}(\exists L_i : \theta_i \wedge p_i))$ for use in the FLL algorithm.

Emerson and Lei consider a general fairness criterion in [13], which is a disjunction of strong fairness conditions. This can be handled by abstracting each disjunct separately and re-forming the disjunction.

For simplicity, the development of the algorithm considered fairness assertions $(\bigwedge i : \mathsf{FG}(p_i) \vee \mathsf{GF}(q_i))$ where p_i and q_i are expressed in terms of the variables of process P_i. In a more general setting, these predicates may be expressed over the local state of more than one process. The analysis method extends easily, with each predicate being abstracted by quantifying out the relevant local variables. Thus, the general abstraction function is $p^\theta \equiv (\exists L : (\bigwedge i : \theta_i) \wedge p)$.

5 Experimental Results

We implemented our method as part of SPLIT [10] – a compositional LTL verifier, and tested it on several parameterized examples which require fairness assumptions. We also compared it with the LTL model checker implemented on top of JTLV [26], and with the model checker NUSMV [4]. The latter, however, is optimized for verifying synchronous systems and even after disabling the conjunctive partitioning the results obtained by it were considerably inferior to those obtained by JTLV and SPLIT. We therefore do not include in this paper the results obtained by NUSMV. The experiments were conducted on a Intel Core 2 Duo 2.4 GHz with 4 GB RAM running 64-bit Linux. Both SPLIT and JTLV were configured to use the CUDD BDD library. We set a timeout of 20 minutes for the experiments.

The experiments test the method on a number of well-known parameterized protocols. These protocols form a good set of benchmarks: they represent succinct models of standard synchronization patterns found in concurrent software; their characteristics (e.g., proof structure and complexity) are well known, making comparisons with other methods easy for the reader. While the descriptions are short, standard model checking

Table 1. Experimental results when assuming only unconditional fairness

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
1	BAKERY	*no-starvation*	3	300K	0.3	2	1.2M	2.5	2	0.9M	1.5
		– Valid –	4	11.6M	93	2	14.6M	52	2	7.4M	17.4
2	MUXSEM	*no-starvation*	5	58K	0.2	1	48K	0.3	1	44K	0.3
		– Invalid –	10	21M	24	2	371K	1.1	2	330K	1
			20	over 20 minutes		2	2.1M	9	2	1.9M	8.3

Table 2. Experimental results when assuming only strong fairness only over P[0]

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
3	MUXSEM	*no-starvation*	5	24K	0	1	61K	0.2	1	38K	0.2
		– Valid –	10	1.2M	3.8	1	259K	0.7	1	142K	0.5
			20	over 20 minutes		1	1.2M	3	1	697K	1.5

f: **array** $[0..N]$ **of boolean initially** $f = 1$

$$P[1] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0\colon \textit{Non-Critical} \\ l_1\colon \textbf{request}\ f[2] \\ l_2\colon \textbf{request}\ f[1] \\ l_3\colon \textit{Critical} \\ l_4\colon \textbf{release}\ f[1] \\ l_5\colon \textbf{release}\ f[2] \end{array}\right] \end{array}\right] \quad \underset{i=2}{\overset{N}{\|}} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0\colon \textit{Non-Critical} \\ l_1\colon \textbf{request}\ f[i] \\ l_2\colon \textbf{request}\ f[i \oplus_N 1] \\ l_3\colon \textit{Critical} \\ l_4\colon \textbf{release}\ f[i \oplus_N 1] \\ l_5\colon \textbf{release}\ f[i] \end{array}\right] \end{array}\right]$$

Fig. 7. Program DINING-PHIL: the dining philosophers

is by no means proportionally easy, as shown by the time-outs in experiments. Both variants (A and B) of the FLL compositional algorithm are examined. In our experiments, variant B has the better performance.

As mentioned in Subsection 4.1, unconditional fairness is sufficient to guarantee various properties in selected protocols. For example, in algorithm BAKERY [18] ensuring individual starvation-freedom, i.e., $\forall i : \mathsf{G}(wait(i) \Rightarrow \mathsf{F}(crit(i)))$, does not require to assume any weak or strong fairness conditions. For other protocols, such as MUXSEM, the same property is not valid when assuming only unconditional fairness, and both JTLV and SPLIT generate valid counter examples when attempting to verify it. The results for checking the eventual access property of $P[1]$ for the two protocols are provided in Table 1. Note that since the property should be over global variables, the location variable of $P[1]$ was exposed to all processes. "N" is the number of processes, "Nodes" is the peak number of BDD nodes generated, "Time" is the run time in seconds, and "Ref." is number of refinements had to be executed by SPLIT. For both examples the run-times are better for SPLIT; for MUXSEM, where counter examples had to be constructed, they are better by several orders of magnitude. Both SPLIT and JTLV required more than 20 minutes for verifying BAKERY for $N = 5$.

Assuming the strong fairness $\mathsf{GF}(P[1].at\_loc_1 \wedge x) \Rightarrow \mathsf{GF}(P[1].at\_loc_2)$ *only* for $P[1]$ is sufficient to prove the correctness of $\mathsf{G}(wait(1) \Rightarrow \mathsf{F}(crit(1)))$ for MUXSEM. Both model checkers indeed validated the property under this condition and the results are provided in Table 2; they are again in favor of our method by a few orders of magnitude.

Most interesting and challenging test cases with respect to fairness are those that require to assume weak or strong fairness conditions for *all* the processes. The first such example is DINING-PHIL (a simple solution to the dining philosophers problem using semaphores), presented in Fig. 7. The eventual access property is valid only when

Table 3. Results for properties that require to assume general fairness over *all* processes

	Example	Property	N	JTLV		SPLIT (A)			SPLIT (B)		
				Nodes	Time	Ref.	Nodes	Time	Ref.	Nodes	Time
4	DINING-PHIL	*no-starvation* – Valid –	8	3M	13	0	1.9M	4	0	1.2M	1.8
			9	9.1M	63	0	4.1M	8.6	0	2.4M	4.3
			10	25M	421	0	8.6M	18	0	5.3M	9.9
5	COND-TERM	*termination* – Valid –	4	91K	0.4	3	389K	1	2	299K	0.8
			6	537K	1.6	3	2.1M	6.7	2	1.6M	5.1
			8	4M	10	3	19M	101	2	11M	75.4
6	MUXSEM-NON-DET	*no-starvation* – Valid –	8	262K	0.6	1	172K	0.5	1	96K	0.4
			12	5.3M	32.6	1	393K	1	1	210K	0.5
			16	over 20 minutes		1	720K	1.8	1	385K	0.9

assuming that $\mathsf{GF}(P[i].at\_loc_1 \wedge f[i]) \Rightarrow \mathsf{GF}(P[i].at\_loc_2)$ and $\mathsf{GF}(P[i].at\_loc_2 \wedge f[i \oplus_N 1]) \Rightarrow \mathsf{GF}(P[i].at\_loc_3)$ for $1 < i \leq N$ and assuming the symmetric conditions for $i = 1$. Namely, for each philosopher, if she can (enabled) infinitely often pick the first fork and subsequently pick the second fork then she should eat Spaghetti infinitely often. Example 4 in Table 3 presents the run-time results for verifying this example, that are again in favor of our method.

$$
\begin{array}{l}
y\colon \textbf{0..M where } y = M \\
\quad \Big\| \; P[i] :: \left[\begin{array}{l} l_0\colon \textbf{while } y \neq 0 \textbf{ do} \\ \quad l_1\colon \left[\begin{array}{l} l_2\colon y := y - 1 \\ \quad \textbf{or} \\ l_3\colon y := min(y+1, M) \end{array}\right] \\ : \; l_4 \end{array}\right] \\
\quad i{=}1
\end{array}^{N}
$$

Fig. 8. COND-TERM

The second example is proving termination of COND-TERM. The protocol is presented in Fig. 8. A process terminates only if the strong fairness condition $\mathsf{GF}(P[i].at\_l_3) \Rightarrow \mathsf{GF}(false)$ is assumed over *all* processes. This condition permits only computations where y is increased a finite number of times. We verified the termination of COND-TERM for $M = 15$. The results are provided as example 5 in Table 3. This time they are in favor of the monolithic model checking as SPLIT requires a number of refinements to prove the property.

The last example that requires to assume general fairness over all the processes is MUX-SEM-NON-DET, presented in figure Fig. 9. This example is a variation of MUXSEM that allows each of the processes to stay non-deterministically, possibly forever, in the critical section. Thus, $\mathsf{G}(wait(1) \Rightarrow \mathsf{F}(crit(1)))$ is valid only when assuming $\bigwedge i : \mathsf{GF}(P[i].at\_l_2) \Rightarrow \mathsf{GF}(P[i].at\_l_3)$. Namely, for each process, if it can (enabled) infinitely often leave the critical section then it should leave it infinitely often. Example 6 in Table 3 presents the run-time results for verifying this example, that are again in favor of our method.

$$
\begin{array}{l}
x : \textbf{boolean where } x = 1 \\
\quad \Big\| _{i=1}^{N} P[i] :: \left[\begin{array}{l} \textbf{loop forever do} \\ \left[\begin{array}{l} l_0 : \mathit{Non\text{-}Critical} \\ l_1 : \textbf{request } x \\ l_2 : \langle \mathit{Critical};\ \textbf{await } (\mathit{false}) \textbf{ or skip} \rangle \\ l_3 : \textbf{release } x \end{array} \right] \end{array} \right]
\end{array}
$$

Fig. 9. MUX-SEM-NON-DET: mutual exclusion with a non-deterministic stay in critical section

1. Find a vector of local assertions, $\theta = (\theta_1, \ldots, \theta_N)$, which meets the split invariance conditions from Figure 1
2. Form a fairness assertion, Ξ, out of the abstract assertions in Φ^θ and the acceptance condition of the Büchi automaton for the negated property. For each i, instantiate the strong fairness proof rule of [25] for the synchronous composition of the automaton and the abstract process P_i^θ, with the fairness assertion Ξ and specification $\mathsf{G}(\mathit{true} \ \Rightarrow \ \mathsf{F}\mathit{false})$.

Fig. 10. Local Proof Rule for LTL properties

6 Deductive Compositional Proofs under Fairness

The LL method was derived in [9] from a proof rule for verifying linear-time properties expressed by a Büchi automaton for their negation. That proof rule has two parts: the first part expresses that θ is a split invariant, while the second part shows that a Büchi accepting state occurs only finitely often on any joint computation of the program and the automaton, using rank functions which are local to each process.

This structure can be modified to accommodate fairness, as shown in Figure 10. The proof rule of [25] is used with the conclusion being false. A valid proof shows the absence of any joint computation which is fair and is an accepting Büchi automaton run. All assertions and rank functions are local by definition. Moreover, as shown in [25], one can generate these components by instrumenting the model checking algorithms used in FLL.

7 Conclusions and Future Work

The algorithm presented here enables fully automated and compositional verification of progress properties under fairness and is, we believe, the first algorithm to do so. It deals with the main difficulty, that of handling the global nature of fairness, by a process of refinement: the fairness assumption is initially weakened relative to a split invariant, and is then strengthened in subsequent iterations until a decisive result is obtained. The algorithm has a simple implementation. Experiments with several parameterized protocols show a clear advantage for the compositional method over the standard non-compositional one.

One aspect that merits further exploration is the choice of counter-example trace for refinement; currently, the algorithm uses whichever trace is provided by the model checking procedure. It would help, for instance, if the trace generation is biased to generate a trace which satisfies as many MUST requirements as possible.

References

1. Alur, R., Henzinger, T.: Reactive modules. In: IEEE LICS (1996)
2. Angluin, D.: Learning regular sets from queries and counterexamples. Inf. Comput. 75(2), 87–106 (1987)
3. Chandy, K.M., Misra, J.: Parallel Program Design: A Foundation. Addison-Wesley, Reading (1988)
4. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NuSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
5. Clarke, E.M., Emerson, E.A., Sistla, A.P.: Automatic verification of finite-state concurrent systems using temporal logic. ACM Transactions on Programming Languages and Systems (TOPLAS) 8(2) (1986)
6. Clarke, E.M., Grumberg, O.: Avoiding the state explosion problem in temporal logic model checking. In: PODC, pp. 294–303 (1987)
7. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: ISSTA, pp. 97–108 (2006)
8. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
9. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
10. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: SPLIT: A Compositional LTL Verifier. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 558–561. Springer, Heidelberg (2010)
11. de Roever, W.-P., de Boer, F., Hannemann, U., Hooman, J., Lakhnech, Y., Poel, M., Zwiers, J.: Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods. Cambridge University Press, Cambridge (2001)
12. Dijkstra, E.W., Scholten, C.S.: Predicate Calculus and Program Semantics. Springer, Heidelberg (1990)
13. Emerson, E.A., Lei, C.-L.: Efficient model checking in fragments of the propositional mu-calculus (extended abstract). In: LICS (1986)
14. Emerson, E.A., Lei, C.-L.: Modalities for model checking: Branching time logic strikes back. Sci. of Comp. Programming 8(3) (1987)
15. Farzan, A., Chen, Y., Clarke, E.M., Tsan, Y., Wang, B.: Extending automated compositional verification to the full class of omega-regular languages. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 2–17. Springer, Heidelberg (2008)
16. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
17. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE, pp. 3–12 (2002)
18. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. ACM Commun. 17(8), 453–455 (1974)
19. Lamport, L.: Proving the correctness of multiprocess programs. IEEE Trans. Software Eng. 3(2) (1977)
20. McMillan, K.L.: A compositional rule for hardware design refinement. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 24–35. Springer, Heidelberg (1997)
21. McMillan, K.L.: Circular compositional reasoning about liveness. In: Pierre, L., Kropf, T. (eds.) CHARME 1999. LNCS, vol. 1703, pp. 342–346. Springer, Heidelberg (1999)

22. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
23. Namjoshi, K.S., Trefler, R.J.: On the completeness of compositional reasoning. In: Emerson, E.A., Sistla, A.P. (eds.) CAV 2000. LNCS, vol. 1855, pp. 139–153. Springer, Heidelberg (2000)
24. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. ACM Commun. 19(5), 279–285 (1976)
25. Pnueli, A., Sa'ar, Y.: All you need is compassion. In: Logozzo, F., Peled, D.A., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 233–247. Springer, Heidelberg (2008)
26. 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), http://jtlv.ysaar.net/
27. Tkachuk, O., Dwyer, M.B., Pasareanu, C.S.: Automated environment generation for software model checking. In: ASE, pp. 116–129 (2003)

SPLIT: A Compositional LTL Verifier

Ariel Cohen[1], Kedar S. Namjoshi[2], and Yaniv Sa'ar[3,⋆]

[1] New York University, New York, NY
arielc@cs.nyu.edu
[2] Bell Labs, Alcatel-Lucent, Murray Hill, NJ
kedar@research.bell-labs.com
[3] Weizmann Institute of Science, Rehovot, Israel
yaniv.saar@weizmann.ac.il

Abstract. This paper describes SPLIT, a compositional verifier for safety and general LTL properties of shared-variable, multi-threaded programs. The foundation is a computation of compact local invariants, one for each process, which are used for constructing a proof for the property. An automatic refinement procedure gradually exposes more local information, until a decisive result (proof/disproof) is obtained.

1 Introduction

Standard model checking algorithms prove safety properties through a reachability computation, computing an inductive assertion (the reachable states) that is defined over the full state vector. They often suffer from the state explosion problem [2]; for concurrent programs, this is manifested as an exponential growth of the state space with increasing number of components.

SPLIT is a new tool for the verification of shared-variable, asynchronous concurrent programs, which ameliorates state explosion through assertional (i.e., state-based) compositional reasoning, based on the classical Owicki-Gries method [13]. The foundation is a construction of a *vector* of local (i.e., per-process) inductive invariants, $\theta = (\theta_1, \theta_2, \ldots, \theta_N)$. The invariants are mutually interference-free—i.e., a move by one process does not violate the local invariant of another. Such a vector is called a *split-invariant*, as the conjunction of its components, $(\bigwedge_i \theta_i)$, is always a *globally inductive* invariant. Locality is enforced by syntactically limiting each process assertion to the variables visible to that process—i.e., the globally shared and process-local variables.

SPLIT implements a number of algorithms; together, they result in a fully automatic compositional model checker for general LTL properties.

1. A simultaneous least fixpoint algorithm [12], which computes the *strongest* split invariant vector (A split invariant is usually weaker than the set of reachable states.)
2. A safety refinement method [4], which achieves completeness by gradually "exposing" local predicates (i.e., encoding them as shared variables)
3. A compositional algorithm which verifies arbitrary LTL properties [5], based on a split invariance computation and a counter-example based refinement scheme

⋆ This research was supported by the John von Neumann Minerva Center for the Development of Reactive Systems at the Weizmann Institute of Science, and by an Advanced Research Grant awarded to David Harel from the European Research Council (ERC) under the European Community's 7th Framework Programme (FP7/2007-2013).

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 558–561, 2010.

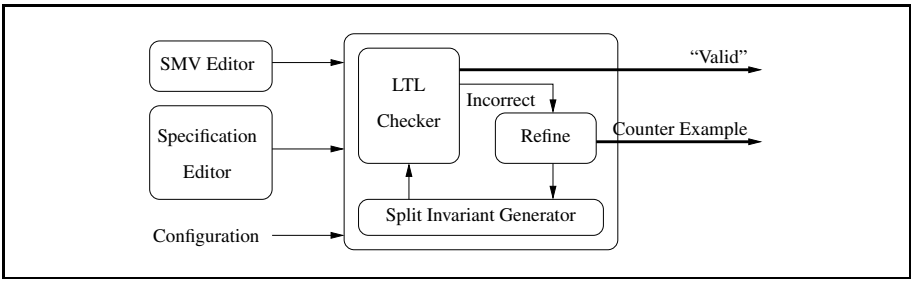


Fig. 1. The architecture of SPLIT

4. A recently developed compositional algorithm [6], for the verification of progress properties under general fairness assumptions

Experimental results support the hypothesis that local reasoning allows verifying significantly larger systems without running into state explosion, and can result in order-of-magnitude improvements in run-time over monolithic model checking. It is interesting that basic local reasoning suffices for the proofs for many protocols, without a need for refinement. In many other cases, a proof/disproof is obtained by exposing a limited amount of local state, validating the basic argument for compositional verification. SPLIT has been used to verify protocols for cache coherence and mutual exclusion. To the best of our knowledge, this is the first tool to implement a fully automated compositional method for both safety and liveness properties.

2 Architecture and Selected Features

SPLIT is built using JTLV [14] – a BDD-based framework for developing temporal verification algorithms. Fig. 1 sketches the architecture of SPLIT. It takes three inputs: an SMV [11] program, an LTL specification, and a configuration. The main part of SPLIT is built up from three components: a unit that generates the split invariant, a verifier for LTL properties, and a unit to compute refinements.

Verification is implemented differently for safety and liveness properties. For a safety property, the algorithm (from [12, 4]) first checks if the split invariant implies the property. If it does, then the property is valid; otherwise, the refinement unit heuristically selects local predicates and "exposes" them. (A local predicate p is exposed by adding an auxiliary shared variable, say x_p, in such a manner that the invariant $(x_p \equiv p)$ is maintained.) Exposing local state strengthens the split invariant in the next iteration; the process is repeated until the property is proved or no additional refinements can be performed. In the latter case, SPLIT generates a valid counter-example trace.

For a liveness property, the algorithm (from [5, 6]) uses the computed split invariant to construct abstract forms of each process. It checks if the liveness property is satisfied by *all* abstract processes, using a standard LTL checker from JTLV. If all checks succeed, the property is valid; otherwise, a counter example trace is extracted. If the trace is spurious, it is used by a refinement procedure to expose local predicates. This process is repeated until either the property is proved or a valid counter-example trace is found.

The user interface for SPLIT allows the user to expose local variables, which can help reduce the number of refinement steps. The counter-examples produced are augmented

Table 1. Characteristic experimental results. (More results are on the tool web page.)

Example	Property	N	JTLV		SPLIT	
			Nodes	Time	Nodes	Time
SEMAPHORE (+COUNT)	mutual exclusion	10	1.2M	10.4	160k	0.3
	– valid –	12	1.8M	440	252k	0.5
PETERSON'S	mutual exclusion	5	6.9M	16	3.7M	8.1
	– valid –	6	91M	509	43.8M	172
BAKERY	mutual exclusion	7	2.9M	65	7.8M	20
	– valid –	8	11M	844	27M	97
SZYMANSKI	mutual exclusion	3	68k	0.1	788k	2.4
	– valid –	4	395k	0.6	3.8M	10
SEMAPHORE	individual starvation-freedom	10	21M	24	371k	1.1
	– Counter example –	20	over 20 minutes		2.1M	9
BAKERY	individual starvation-freedom	3	300k	0.3	1.2M	2.5
	– Valid –	4	11.6M	93	14.6M	52
DINING-PHIL	individual starvation-freedom	9	9.1M	63	4.1M	8.6
	– Valid –	10	25M	421	8.6M	18

with refinement predicates that express the changes to the state. SPLIT is implemented in about 9000 lines of Java, of which at least half is for the user interface. It relies on standard BDD libraries written in C and Java. More information, including a collection of examples, can be found at `http://split.ysaar.net/`.

3 Experimental Results

We have used SPLIT to verify safety and liveness properties for a number of multi-threaded protocols for mutual exclusion and cache coherence. Table 1 presents characteristic results of comparing SPLIT with the (monolithic) LTL model checker in JTLV. Both were configured to use the CUDD BDD library. In the table, "N" is the number of processes, "Nodes" is the peak number of BDD nodes generated, and "Time" is the runtime in seconds.

In nearly all cases (SZYMANSKI being the exception) SPLIT obtained better run-times, sometimes showing as much as one or two orders of magnitude improvement. Improvement in memory consumption, which is proportional to the number of peak BDD nodes, is not as clear-cut: for BAKERY, for which it obtains better run-times, SPLIT requires more memory. SPLIT was also able to verify much larger systems than the monolithic model checker; for instance, it proves SEMAPHORE for $N = 64$ where JTLV ran out of memory already for $N = 24$. The performance of NUSMV [1] on most of these examples was inferior to that of JTLV and SPLIT even after disabling the conjunctive partitioning. This appears to be because NUSMV is optimized for verifying synchronous systems and we therefore do not include the results obtained by it.

4 Related Work and Conclusions

SPLIT mechanizes assertional (i.e., state-predicate based) compositional reasoning in the style of the seminal Owicki-Gries proof method. Thread-modular reasoning [8]

computes the strongest split invariant with an explicit-state algorithm, but it does not include a refinement step and is therefore incomplete. An alternative automated compositional method is based on behavioral (i.e., path-based) reasoning, and uses automaton learning algorithms [10,9]. Experimental results with this method have been mixed [3]: in many cases, monolithic verification is faster; mostly due to exponential (on the number of variables) alphabet complexity, and partly due to aiming for a deterministic representation of the assumption. Assertional reasoning has a simple implementation, even for the analysis of general LTL properties, and the experiments with SPLIT show a clear advantage over monolithic verification on a number of protocols.

There are several potential improvements and extensions being investigated in current work. One focus is on coupling counter-example generation with refinement; the current implementation uses whichever trace is provided by the JTLV model checker. Another focus is on parallel and distributed implementations [7], as the compositional reasoning calculations can be easily parallelized.

References

1. Cimatti, A., Clarke, E.M., Giunchiglia, F., Roveri, M.: NUSMV: a new symbolic model verifier. In: Halbwachs, N., Peled, D.A. (eds.) CAV 1999. LNCS, vol. 1633, pp. 495–499. Springer, Heidelberg (1999)
2. Clarke, E.M., Grumberg, O.: Avoiding the state explosion problem in temporal logic model checking. In: PODC, pp. 294–303 (1987)
3. Cobleigh, J.M., Avrunin, G.S., Clarke, L.A.: Breaking up is hard to do: an investigation of decomposition for assume-guarantee reasoning. In: ISSTA, pp. 97–108 (2006)
4. Cohen, A., Namjoshi, K.S.: Local proofs for global safety properties. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 55–67. Springer, Heidelberg (2007)
5. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
6. Cohen, A., Namjoshi, K.S., Sa'ar, Y.: A dash of fairness for compositional reasoning. In: Touili, T., Cook, B., Jackson, P. (eds.) CAV 2010. LNCS, vol. 6174, pp. 543–557. Springer, Heidelberg (2010)
7. Cohen, A., Namjoshi, K.S., Sa'ar, Y., Zuck, L.D.: Symbolic model checking on multi-core processors. Technical report, Bell Laboratories (2009)
8. Flanagan, C., Qadeer, S.: Thread-modular model checking. In: Ball, T., Rajamani, S.K. (eds.) SPIN 2003. LNCS, vol. 2648, pp. 213–224. Springer, Heidelberg (2003)
9. Giannakopoulou, D., Pasareanu, C.S.: Learning-based assume-guarantee verification (tool paper). In: Godefroid, P. (ed.) SPIN 2005. LNCS, vol. 3639, pp. 282–287. Springer, Heidelberg (2005)
10. Giannakopoulou, D., Pasareanu, C.S., Barringer, H.: Assumption generation for software component verification. In: ASE, pp. 3–12 (2002)
11. McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)
12. Namjoshi, K.S.: Symmetry and completeness in the analysis of parameterized systems. In: Cook, B., Podelski, A. (eds.) VMCAI 2007. LNCS, vol. 4349, pp. 299–313. Springer, Heidelberg (2007)
13. Owicki, S.S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. ACM Commun. 19(5), 279–285 (1976)
14. 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), http://jtlv.ysaar.net/

A Model Checker for AADL*

Marco Bozzano[2], Alessandro Cimatti[2], Joost-Pieter Katoen[1],
Viet Yen Nguyen[1], Thomas Noll[1], Marco Roveri[2], and Ralf Wimmer[3]

[1] RWTH Aachen University, Germany
[2] Fondazione Bruno Kessler, Trento, Italy
[3] Albert-Ludwigs-University Freiburg, Germany

Abstract. We present a graphical toolset for verifying AADL models, which are gaining widespread acceptance in aerospace, automobile and avionics industries for comprehensively specifying safety-critical systems by capturing functional, probabilistic and hybrid aspects. Analyses are implemented on top of mature model checking tools and range from requirements validation to functional verification, safety assessment via automatic derivation of FMEA tables and dynamic fault trees, to performability evaluation, and diagnosability analysis. The toolset is currently being applied to several case studies by a major industrial developer of aerospace systems.

1 Introduction

System-level languages like Architecture Analysis and Design Language (AADL) and SysML are increasingly adopted by industry for designing new safety-critical systems. Their added advantage is that they enable the system designer to capture the elusive interaction between hardware and software. In particular nominal and degraded modes of operation, the propagation of faults between subsystems, and the mechanisms for the system to recover from them are essential to a comprehensive system-level design.

As part of the COMPASS project [1] (Correctness, Modelling and Performance of Aerospace Systems) we developed a formal semantics for AADL (briefly discussed in Sect. 2) that incorporates functional, probabilistic and hybrid aspects [5]. This is fundamental for tool-supported formal verification. Over the past two years we have built a graphical toolset, called the COMPASS platform (see Sect. 3), supporting AADL and based on model checking techniques to verify them. The tool is currently being applied to several case studies by a major industrial developer of aerospace systems [4].

2 Specification Language

To make AADL amenable to formal verification, we have cut out its superfluous features and added support for hybrid aspects. The tool's resulting input language thus follows the component-based paradigm. It supports both software (e.g., processes and threads) and hardware components (e.g., memories and processors) as first-class objects. Each component is given by its type, describing the interface, and its implementation, describing the interactions via a finite state automaton. Sets of interacting

* Funded by ESA/ESTEC under Contract No. 21171/07/NL/JD.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 562–565, 2010.

components can be grouped into composite components, enabling the modeler to manage the system's complexity by introducing a component *hierarchy*. Communication is achieved via exchange of messages on event ports, in a rendez-vous manner. Moreover, components may exchange data through typed data ports (e.g., bool, integer and real data types). Timed and hybrid behavior can be expressed by means of real-valued variables with (linear) time-dependent dynamics.

The resulting hierarchical system model, also referred to as *nominal model*, describes the system behavior under normal operation. This is complemented by an *error model* which expresses how the system can fail. Moreover, a subset of the nominal components may be designated as dealing with error diagnosis and recovery; they are referred to as FDIR (Fault Detection, Identification and Recovery). The error model expresses how faults may affect normal operation and may lead the system into a degraded mode of operation. It is modeled as a (probabilistic) finite state automaton, where transitions may occur due to error events which may be annotated with a rate that indicates the expected number of occurrences per time unit. Transitions can also occur because of error propagations from other components. The nominal and error models are linked through a so-called *fault injection*. A fault injection expresses the effect of the occurrence of the corresponding error on the nominal model. Multiple fault injections are possible. The process of integrating the nominal models with the error models and the fault injections, is called *model extension* [8]. Finally, in order to enable modeling of partial observability and analysis of FDIR components, our language allows the modeler to explicitly define a set of observables.

We refer to [5,6,7] for a more detailed description of the tool's input language, a discussion of the similarities and extensions with respect to AADL, and a simple example (a processor failover system). In particular, [5] presents a formal semantics for all the language constructs, based on networks of event-data automata (NEDA).

3 Toolset

The COMPASS platform [7] is an integrated toolchain, based on state-of-the-art tools and symbolic model checking techniques, for verification and validation of AADL models. It builds upon the NuSMV [16] symbolic model checker, the MRMC [15] probabilistic model checker, the Sigref [18] bisimulation minimization tool, and the RAT [17] requirements analysis tool. The architecture of the tool is sketched in Figure 1. It refers to two inputs, namely the SLIM model (our extended variant of AADL) and property patterns. The latter describe properties, expressed in the user-friendly patterns by Grunske [14] and Dwyer [12], which are converted to respectively CSL [3], LTL and CTL formulae. These inputs are processed into the lower-level formalisms of NuSMV and MRMC. A set of visualizers transforms the output (like counterexample traces and fault trees) back to the user. Feature-wise, the toolset supports the following analyses.

Requirements Validation, implemented by RAT, focuses on assessing the quality of a set of requirements with respect to the user expectations, before a model of the actual system is built. It is possible to check that a set of properties is logically consistent, and that it is strict enough but not too strict, by checking for compatibility with a set of possibilities, and for logical consequence of a set of assertions.

Functional Verification comprises random and user-guided simulation, deadlock detection, and verification of functional properties via model checking. The result can be a statement that a property holds, or a counterexample or witness trace, in case the property is refuted or for simulation. This analysis is based on NuSMV, which supports BDD-based, SAT-based, and (for hybrid systems) SMT-based model checking.

Safety Analysis comprises traditional techniques for hazard analysis, such as (Dynamic) Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA), that are used to assess system behavior in presence of faults. FTA constructs all possible chains of basic faults, represented as a tree, that may be responsible for an undesired behavior. FMEA is similar, but starts from a set of faults and analyzes the impact on a set of properties.

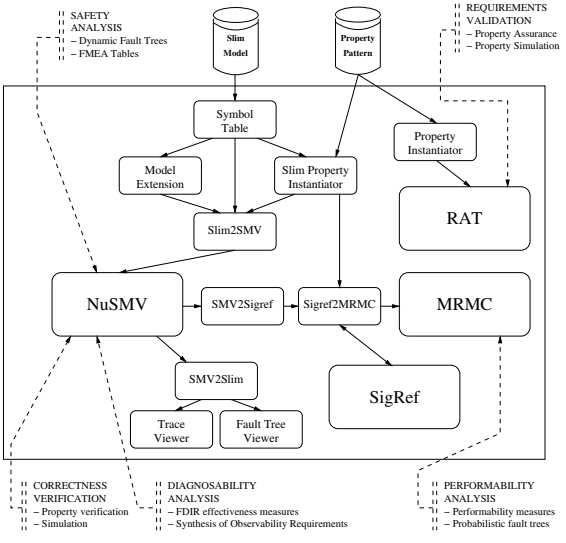


Fig. 1. Architecture of the toolset

Diagnosability Analysis checks whether a system is diagnosable with respect to a user-specified property, that is, whether an ideal diagnoser has enough observations to identify the set of causes of a specific faulty behavior.

Fault Detection, Isolation and Recovery (FDIR) focus on, respectively, verifying whether a given fault can be properly detected, isolated and recovered.

Performance Evaluation computes, using a probabilistic property, system performance under degraded operations. It furthermore includes the evaluation of fault trees by computing the probability of the top event. These analyses are based on MRMC.

The toolset is being extensively evaluated on a set of industrial-size case-studies [4]. Moreover, it is being tuned to achieve optimal performance. Preliminary experiments indicate that the choice of the verification technique (e.g., BDD-based versus SAT-based) may be important to achieve a good performance and scalability for the different types of analyses. The details of those experiments can be found on the project's website [1].

4 Related Work and Conclusions

In this paper we have presented a comprehensive toolset for the verification and validation of AADL models. The toolset supports several analyses ranging from requirements validation to functional verification, safety analysis, diagnosability and performance evaluation. It is available under an open source license via the COMPASS website [1].

Several tools have been developed to analyse AADL specifications, however none of them provides support for all analysis described in this paper. We mention AADL2BIP [11], which translates AADL models into a formalism called BIP, and is able to perform simulation and deadlock detection. ADeS [2] is a software tool to simulate the behaviour of system architectures described in AADL. This tool is mainly a simulator that allows for the evaluation and analysis of system behavior, with no formal analysis underneath. Finally, Cheddar [10] and the Furness^{TM} Toolset [13] support only schedulability analysis.

Due to constraints on paper-length, this paper only describes an overview of the COMPASS toolset. An in-depth discussion of the toolset, the formal semantics of AADL and the relation to other works can be found in our journal paper [9].

References

1. The COMPASS project, http://compass.informatik.rwth-aachen.de/
2. ADeS, a simulator for AADL, http://www.axlog.fr/aadl/ades_en.html
3. Baier, C., Haverkort, B., Hermanns, H., Katoen, J.-P.: Model-checking algorithms for continuous-time Markov chains. IEEE Trans. on Soft. Eng. 29(6), 524–541 (2003)
4. Bozzano, M., Cavada, R., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Olive, X.: Formal Verification and Validation of AADL Models. In: Proc. ERTS 2010 (to be published, 2010)
5. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Codesign of Dependable Systems: A Component-Based Modelling Language. In: Proc. MEMOCODE'09, pp. 121–130. IEEE, Los Alamitos (2009)
6. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Model-based codesign of critical embedded systems. In: Proc. ACES-MB'09, pp. 87–91 (2009)
7. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In: Buth, B., Rabe, G., Seyfarth, T. (eds.) SAFECOMP 2009. LNCS, vol. 5775, pp. 173–186. Springer, Heidelberg (2009)
8. Bozzano, M., Villafiorita, A.: The FSAP/NuSMV-SA Safety Analysis Platform. Int. J. on Software Tools for Technology Transfer 9(1), 5–24 (2007)
9. Bozzano, M., Cimatti, A., Katoen, J.-P., Nguyen, V.Y., Noll, T., Roveri, M.: Safety, dependability, and performance analysis of extended AADL models. The Computer Journal (March 2010) doi: 10.1093/com
10. Cheddar: a free real time scheduling tool, https://wiki.sei.cmu.edu/aadl/index.php/Cheddar
11. Chkouri, M.Y., Robert, A., Bozga, M., Sifakis, J.: Translating AADL into BIP – application to the verification of real-time systems. In: Proc. ACES-MB'08, pp. 39–53. Springer, Heidelberg (2008)
12. Dwyer, M., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proc. ICSE'99, pp. 411–420. IEEE, Los Alamitos (1999)
13. The Furness^{TM} Toolset, http://www.furnesstoolset.com/
14. Grunske, L.: Specification patterns for probabilistic quality properties. In: Schäfer, W., Dwyer, M.B., Gruhn, V. (eds.) ICSE, pp. 31–40. ACM, New York (2008)
15. MRMC – Markov Reward Model Checker, http://www.mrmc-tool.org/
16. The NuSMV Model Checker, http://nusmv.fbk.eu
17. RAT – Requirements Analysis Tool, http://rat.fbk.eu
18. Wimmer, R., Herbstritt, M., Hermanns, H., Strampp, K., Becker, B.: Sigref – A Symbolic Bisimulation Tool Box. In: Graf, S., Zhang, W. (eds.) ATVA 2006. LNCS, vol. 4218, pp. 477–492. Springer, Heidelberg (2006)

PESSOA: A Tool for Embedded Controller Synthesis*

Manuel Mazo Jr.[1], Anna Davitian[2], and Paulo Tabuada[1]

[1] CyPhyLab, University of California, Los Angeles CA 90095, USA
[2] Aerovironment Inc., Monrovia, CA 91016, USA

Abstract. In this paper we present Pessoa, a tool for the synthesis of correct-by-design embedded control software. Pessoa relies on recent results on approximate abstractions of control systems to reduce the synthesis of control software to the synthesis of reactive controllers for finite-state models. We describe the capabilities of Pessoa and illustrate them through an example.

1 Introduction

The synthesis of embedded control software is a challenging task due to the complex interactions between the physical and computational processes involved in embedded applications. The tool introduced in this paper, named Pessoa[1], automatically synthesizes embedded controllers enforcing several temporal logic specifications on physical systems. The controllers synthesized by Pessoa are described by Binary Decision Diagrams (BDD's) [Weg00], which have been shown to be adequate for the automatic generation of hardware [bloem07] or software [Be99] implementations. The tool Pessoa illustrates the correct-by-design approach to the synthesis of embedded control software by generating BDDs, describing the control software, from a formal specification.

Most of the tools available for hybrid systems such as Ariadne [Ari], PHAVer [PHA], KeYmaera [KeY], Checkmate [Che], and HybridSAL [Hyba], focus on verification problems. Tools for the synthesis of controllers are more recent and include LTLCon [LTL] for linear control systems and the Hybrid Toolbox [Hybb] for piece-wise affine hybrid systems. What sets Pessoa apart from the existing synthesis tools is the nature of the abstractions (approximate simulations and bisimulations) and the classes of systems admitting such abstractions (linear, nonlinear, and switched [Tab09]). Although Pessoa does not support nonlinear and switched systems natively, they can already be handled as illustrated in the examples in [Pes10].

* This work was partially supported by the NSF awards 0717188, 0820061, and 0834771.

[1] Pessoa Version 1.0 can be freely downloaded from `http://www.cyphylab.ee.ucla.edu/pessoa/`

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 566–569, 2010.

2 Formal Models for Software and Control

Pessoa uses the following notion of system allowing to describe both software and control systems under the same paradigm.

Definition 1. *A* system $S = (X, X_0, U, \longrightarrow)$ *is a tuple consisting of:*

- *a set of* states X*;*
- *a set of* initial states $X_0 \subseteq X$*;*
- *a set of* inputs U*;*
- *a* transition relation $\longrightarrow \subseteq X \times U \times X$*;*

System S is said to be finite when X has finite cardinality and metric when X is equipped with a metric $\mathbf{d} : X \times X \to \mathbb{R}_0^+$.

The "dynamics" of a system is described by the transition relation: existence of a transition $(x, u, x') \in \longrightarrow$ entails that upon the reception of input u at state x, system S evolves to state x'. In [Tab09] it is shown how systems of this form can represent both software and control systems modeling physical processes. While software models naturally lead to finite systems, obtaining models for control systems leading to finite systems requires of some abstraction techniques. Informally, a control system Σ is a differential equation of the form $\dot{\xi} = f(\xi, \upsilon)$, where $\xi(t)$ denotes the state of the system at time t, $\upsilon(t)$ the controlled input, and $\dot{\xi}$ denotes the time derivative of ξ. By adequately discretizing the state space of the differential equation and the input space, with discretization steps η and μ respectively, and by selecting an adequate sampling time τ, it has been shown in [Tab09] and references therein, that useful finite abstractions $S(\Sigma)$ for Σ can be obtained. Such finite abstractions can be related to the control system through a generalization of the notion of *alternating simulation relation* named *alternating approximate simulation relation* [Tab09]. The existence of such relations reduces the synthesis of controllers for Σ to the synthesis of reactive controllers for the finite abstraction $S(\Sigma)$.

3 Pessoa Functionalities

Pessoa is a Matlab Toolbox. Although the core algorithms have been coded in C, the main functionalities are available through the Matlab command line. All the systems and sets manipulated by Pessoa are represented symbolically using Reduced Ordered Binary Decision Diagrams (ROBDDs) supported by the CUDD library [CUD]. Pessoa Version 1.0 offers three main functionalities:

1. the construction of finite abstractions of linear control systems;
2. the synthesis of reactive controllers for simple temporal logic specifications;
3. refinement of reactive controllers to a ROBDD description of the control software used to simulate the closed-loop behavior in Simulink.

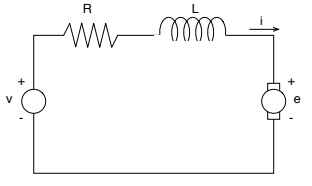
Pessoa currently supports the synthesis of controllers enforcing four kinds of linear temporal logic specifications defined using a target set $Z \subseteq X$ and a constraint set $W \subseteq X$:

1. **Stay:** $\Box\varphi_Z$ where φ_Z is the predicate defining the set Z;
2. **Reach:** $\Diamond\varphi_Z$;
3. **Reach and Stay:** $\Diamond\Box\varphi_Z$;
4. **Reach and Stay (in Z) while Stay (in W):** $\Diamond\Box\varphi_Z \wedge \Box\varphi_W$ where φ_W is the predicate defining the set W.

For lack of space it is not possible to provide further details on how the abstraction and synthesis algorithms are implemented. Such details can be found in [Pes10].

4 Example

The example that we consider consists in regulating the velocity of a DC motor. The electric circuit driving the DC motor and the two linear differential equations defining the dynamics Σ of this system are shown in Figure 1. The variable x_1 describes the angular velocity of the motor, the variable x_2 describes the current i through the inductor, and the variable u represents the source voltage v that is treated as an input. The model parameters take the following values in international system units: $R = 500 \times 10^{-3}$ (resistance), $L = 1500 \times 10^{-6}$ (inductance), $J = 250 \times 10^{-6}$ (moment of inertia), $B = 100 \times 10^{-6}$ (viscous friction coefficient) and $k = 50 \times 10^{-3}$ (torque constant).



$$\dot{x}_1 = -\frac{B}{J}x_1 + \frac{k}{J}x_2 \qquad (1)$$

$$\dot{x}_2 = -\frac{k}{L}x_1 - \frac{R}{L}x_2 + \frac{1}{L}u. \qquad (2)$$

Fig. 1. DC motor circuit and equations describing its dynamics

The control objective is to regulate the velocity around 20 rad/s. In practical implementations the DC motor is connected to a constant voltage source through an H-bridge. By opening and closing the switches in the H-bridge we can only choose three different values for the voltage: -10V, 0V, and 10V. In order to synthesize a controller under these input constraints we define the input space to be $\mathsf{U} = [-10, 10]$ and set the input quantization to $\mu = 10$. The time quantization is set to $\tau = 0.0001$ and the space quantization to $\eta = 0.05$. The resulting abstraction is computed in 17 minutes on a MacBook Pro with a 2.26 GHz Intel Core 2 Duo processor and 2GB of memory.

The target set is selected to be $Z = [19.5, 20.5] \times [-0.7, 0.7]$ so as to obtain a current ripple no larger than 0.7 Amperes around 0 while reaching the desired angular velocity. Moreover, by introducing the constraint set $W = [-1, 30] \times [-3, 3]$ the peak current is limited to 3 Amperes. We synthesize a controller enforcing the "reach and stay while stay" specification in 108 seconds. The closed-loop simulation results in Figure 2 show that the target set is reached while the current ripple and peak values limitations are respected. For a more detailed version of this example and other illustrative examples of Pessoa capabilities we refer the reader to [Pes10].

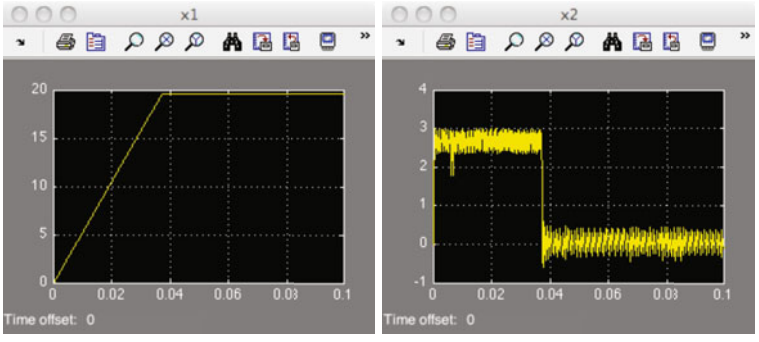


Fig. 2. Evolution of velocity (left) and current (right)

5 Conclusions and Future Work

Pessoa can be used to compute finite abstractions of continuous control systems, synthesize reactive controllers, and refine the synthesized controllers to Simulink blocks. Pessoa is currently being improved by extending its scope to natively support the abstraction of non-linear and switched continuous dynamics, allow full LTL specifications, multi-resolution quantization of the input and state spaces of the control system, support quantitative control objectives, and provide automatic code generation.

References

[Ari] Ariadne, http://trac.parades.rm.cnr.it/ariadne/

[Be99] Balarin, F., et al.: Synthesis of Software Programs for Embedded Control Applications. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 18(6), 834–849 (1999)

[bloem07] Bloem, R., et al.: Specify, Compile, Run: Hardware from PSL. Electron. Notes Theor. Comput. Sci. 190(4), 3–16 (2007)

[Che] Checkmate, http://www.ece.cmu.edu/~webk/checkmate/

[CUD] Cudd, http://vlsi.colorado.edu/~fabio/CUDD/

[Hyba] Hybridsal, http://sal.csl.sri.com/hybridsal/

[Hybb] Hybrid Toolbox, http://www.dii.unisi.it/hybrid/toolbox

[KeY] Keymaera, http://symbolaris.com/info/KeYmaera.html

[LTL] LTLCon, http://iasi.bu.edu/~software/LTL-control.htm

[Pes10] Pessoa internal report UCLA-CyPhyLab-2010-01 (January 2010), http://www.cyphylab.ee.ucla.edu/Home/publications

[PHA] Phaver, http://www-artist.imag.fr/~frehse/phaver_web/index.html

[Tab09] Tabuada, P.: Verification and Control of Hybrid Systems. Springer, Heidelberg (2009)

[Weg00] Wegener, I.: Branching Programs and Binary Decision Diagrams - Theory and Applications. SIAM Monographs on Discrete Mathematics and Applications (2000)

On Array Theory of Bounded Elements*

Min Zhou[1,2,3], Fei He[1,2,3], Bow-Yaw Wang[4], and Ming Gu[1,2,3]

[1] School of Software, Tsinghua University, Beijing 100084, China
[2] Tsinghua National Laboratory for Information Science and Technology
[3] Key Laboratory for Information System Security, MOE, China
[4] INRIA, France, Tsinghua University, China, and Academia Sinica, Taiwan

Abstract. We investigate a first-order array theory of bounded elements. By reducing to weak second-order logic with one successor (WS1S), we show that the proposed array theory is decidable. Finally, two natural extensions to the new theory are shown to be undecidable.

1 Introduction

Consider the following pseudo code for simplified bucket sort:

Input a : `array of` $[0..255]$
`for` $i \leftarrow 0$ `to` 255 `do` $b[i] \leftarrow 0$;
`for` $i \leftarrow 0$ `to` $|a| - 1$ `do` $b[a[i]] \leftarrow b[a[i]] + 1$;
$k \leftarrow 0$;
`for` $i \leftarrow 0$ `to` 255 `do`
  `for` $j \leftarrow 1$ `to` $b[i]$ `do`
    $a[k] \leftarrow i$; $k \leftarrow k + 1$;

In the pseudo code, a is an array of bytes and $|a|$ denotes the size of the array. The array b consists of *buckets*. The v-th bucket records the number of elements in a with value v. Essentially, the pseudo code sorts the input by counting. Since each element of the array a is accessed a constant number of times, the complexity of bucket sort algorithm is linear in the size of the input array.

Suppose we would like to specify properties about the pseudo code. After examining the algorithm, we see that the v-th bucket is positive if and only if there is an index i such that $a[i] = v$. Or, more formally,

$$\forall v \exists i.(b[v] > 0 \leftrightarrow a[i] = v)$$

must hold at the end of the program. This gives a formula in the array logic.

* This work was supported in part by the Chinese National 973 Plan under grant No. 2010CB328003, the NSF of China under grants No. 60635020, 60903030 and 90718039, the National Science Council of Taiwan projects No. NSC97-2221-E-001-003-MY3, NSC97-2221-E-001-006-MY3, the FORMES Project within LIAMA Consortium, and the French ANR project SIVES ANR-08-BLAN-0326-01.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 570–584, 2010.

Array theory in its most general form is undecidable. Several authors have investigated various decidable fragments of the theory. In [12,2,13], arrays are modeled as uninterpreted functions. Existing decision procedures for uninterpreted functions can decide the satisfiability of formulae in array theory. In [4,1], counter automata are used. Satisfiability of array formulae is reduced to the reachability problem of counter automata. In both approaches, their decidable fragments are very restrictive. Arbitrary quantification induces undecidability, and few of the theory fragments above allow nested reads. Particularly, the sample formula given above does not belong to the decidable fragments of these theories.

We consider the theory of arrays with bounded elements in this paper. In our theory, an array can have an arbitrary but finite size. Its elements, however, must be bounded. Quantification is allowed in our first-order theory. Indices, for instance, can be quantified arbitrarily. We show that the first-order theory of arrays with bounded elements is decidable. With this theory, programmers can specify many program properties which cannot be verified by existing fragments. Particularly, the sample formula given above can be solved.

In our theory, bounded elements reflect data storage format in physical world. Arrays with bounded elements can express most program properties naturally. Due to memory limitation, all data types are actually stored in the physical world with a bounded size. For example, variables of the `int` type are actually stored in most of systems as 32-bit integers. Our array theory of bounded elements is surely a realistic abstraction of the array data type.

Adopting bounded elements moreover relieve us from the imposed restriction of decidable fragments in previous theories. In the array theory of bounded elements, arbitrarily quantified first-order formulae are decidable. Nested reads do not induce undecidability either. Both can be freely used in array formulae without incurring computability issues.

In addition to its generality, another significant advantage of adopting the array theory of bounded elements is its simplicity. It is almost standard to establish the decidability result by reducing it to WS1S. Since decision procedures for WS1S are publicly available (for example, MONA [8]), our reduction immediately gives a decision procedure for the array theory of bounded elements.

We also discuss two natural extensions to the array theory of bounded elements. Either unbounded elements or linear arithmetic on indices would introduce undecidability to our theory. Arrays with unbounded elements subsume previous array theories and hence are undecidable. For arrays of bounded elements, the undecidability result of linear arithmetic on array indices is somewhat surprising. We show that multiplication is expressible with linear arithmetic and arrays of bounded elements. Hence linear arithmetic would make the array theory of bounded elements undecidable.

Related Works. In [13], a quantifier-free extensional multi-dimensional array theory is considered. The author employs a variant of congruence closure algorithm to check the satisfiability. Other quantifier-free array theory fragments are considered in [12] and [14].

In [2], a more expressive logic is presented. The author defined a $\exists^*\forall^*$ fragment of array theory. The index theory is alike to Presburger arithmetic, but addition is only allowed on the existential quantified index variables. The authors present a mechanism to instantiate universal quantifiers by replacing the quantified index variables with each term in the index set. Then arrays are treated as uninterpreted functions. It is shown that allowing nested reads or general Presburger arithmetic over universally quantified index variables (even just $i + 1$) induces undecidability.

An automata based approach is presented in [4]. The authors define an specialized array theory SIL which allows formulas like $\forall i.\phi(i) \rightarrow \gamma(i)$ where i is the only index variable in $\gamma(i)$, and disjunction is forbidden in $\gamma(i)$. The same as in [2], quantifiers can be used only in the form of $\exists^*\forall^*$. An SIL formula is translated into a flatten counter automaton, then the satisfiability is checked by testing if the language of automaton is empty. Furthermore, in [1], they encode pieces of program into counter automata. Although SIL is expressive, some restrictions on the structure of formulas are unnatural. Nested reads are not allowed, neither.

WS1S is a simple but expressive logic. An example where integer arrays are broken to bits and expressed as finite sets in WS1S is known in [7]. We employ and further develop this idea in this paper. Furthermore, solving decidability problem by reducing to WS1S is not uncommon. In [11], a decision procedure for bit-vectors by mapping a bit-vector to a set in WS1S is presented.

This paper is organized as follows: First, we give the preliminaries in Sect. 2. Then in Sect. 3, the array theory UABE is presented. In Sect. 4, we show UABE is decidable by a reduction to WS1S. In Sect. 5 we show that extending UABE with unbounded integer element theory or with addition on index variables incurs undecidability. Examples are given in Sect. 6, followed by conclusion.

2 Preliminaries

2.1 WS1S

WS1S [3] is short for Weak Second-order logic with One Successor. It is a second order logic with the signature $\{=, \in, \mathcal{S}\}$. There are both first- and second-order variables in WS1S. Use $\mathcal{V}_1, \mathcal{V}_2$ to represent them respectively. The syntax is defined as:

$$\phi ::= (p = \mathcal{S}(q)) \mid p \in X \mid \neg\phi \mid \phi \vee \phi \mid \exists p.\phi \mid \exists X.\phi, \qquad p, q \in \mathcal{V}_1, X \in \mathcal{V}_2$$

An interpretation is an assignment on variables. First-order variables are interpreted over $\mathbb{N}$ while second-order variables are interpreted over *finite sets* of $\mathbb{N}$. $\mathcal{S}$ is the successor function on $\mathbb{N}$. $\in$ is the membership relation on $\mathbb{N} \times \mathscr{P}(\mathbb{N})$. Given an interpretation σ, the semantics of WS1S is defined as:

$$
\begin{array}{ll}
\sigma \models (p = \mathcal{S}(q)) & \iff \sigma(p) = \sigma(q) + 1 \\
\sigma \models p \in X & \iff \sigma(p) \in \sigma(X) \\
\sigma \models \neg\phi & \iff \sigma \not\models \phi \\
\sigma \models \phi_1 \vee \phi_2 & \iff \sigma \models \phi_1 \text{ or } \sigma \models \phi_2 \\
\sigma \models \exists p.\phi & \iff \sigma[p \leftarrow i] \models \phi, \text{ for some } i \in \mathbb{N} \\
\sigma \models \exists X.\phi & \iff \sigma[X \leftarrow M] \models \phi, \text{ for some } \textit{finite set } M \subseteq \mathbb{N}
\end{array}
$$

Based on the syntax declared above, other predicates such as "$<$", "$\subseteq$" can be defined as: $(p < q) \iff \forall X.(q \in X \wedge \forall r.(\mathcal{S}(r) \in X \rightarrow r \in X) \rightarrow p \in X)$ and $(X \subseteq Y) \iff \forall p.(p \in X \rightarrow p \in Y)$.

There is a correspondence between WS1S formulas and regular languages. Based on this fact, some verification tools such as MONA [8,6] are developed. Given a WS1S formula, MONA is able to give a `valid`, `satisfiable` or `unsatisfiable` answer as well as a satisfying example or a counter-example if any.

We mention another decidable logic named S1S. The difference between S1S and WS1S is: in S1S, second order variables are interpreted as *infinite* sets while in WS1S are interpreted as *finite* sets. S1S corresponds to Büchi Automata. Büchi proved in 1960 and 1961 that both WS1S and S1S are decidable[3].

2.2 Terminology

In this paper, the domains of arrays, array indices and array elements are denoted by $\mathbb{A}$, $\mathbb{N}$, and $\mathbb{Z}_n$ respectively, where $\mathbb{A} = \mathbb{Z}_n^*$, $\mathbb{N} = \{0, 1, 2, \ldots\}$ and $\mathbb{Z}_n = \{0, 1, \ldots, 2^n - 1\}$. To simplify the discussion, we assume $\mathbb{Z}_n$ contains only non-negative values. This is not a limitation of our theory, we will show it is intuitive to extend $\mathbb{Z}_n$ to a more general domain.

We use $Var(\lambda)$ to denote the set of variables whose type is λ, where λ can be $\mathbb{N}$, $\mathbb{Z}_n$ or $\mathbb{A}$. We use Var to denote the set of all variables, i.e. $Var = Var(\mathbb{N}) \cup Var(\mathbb{Z}_n) \cup Var(\mathbb{A})$. Throughout the paper, $i, j, k \in Var(\mathbb{N})$; $x, y, z \in Var(\mathbb{Z}_n)$; $a, b \in Var(\mathbb{A})$; c a constant in $\mathbb{N}$; and l a constant in $\mathbb{Z}_n$. Boolean values are expressed as 0 and 1.

Given an array variable a, we use $a[i]$ to denote the *array read* which returns the i-th element of a, and $a\{i \leftarrow x\}$ the *array write* which returns a new array that is obtained by replacing the i-th element in a with x.

3 The Theory of UABE

In our array theory, the index type is not bounded while the element type is bounded, so we call it Unbounded Array with Bounded Element (UABE). It consists of:

- the index theory $T_{\mathbb{N}}$: with domain of $\mathbb{N}$, and signature of $\{\mathcal{S}, <, =\}$;
- the element theory $T_{\mathbb{Z}_n}$: with domain of $\mathbb{Z}_n$, and signature of $\{+_n, <_n, =_n\}$;
- and one-dimensional extensional array theory with the "*read-over-write*" axiom [10]:

$$
\begin{array}{rl}
\forall a, i, j, x. & (i = j) \rightarrow a\{i \leftarrow x\}[j] = x \\
& \wedge\ (i \neq j) \rightarrow a\{i \leftarrow x\}[j] = a[j]
\end{array}
$$

Table 1. Syntax of UABE

Identifier	Definition	Remarks
$\sim$	$\in \{=, <\}$	comparisons on $\mathbb{N}$
$\sim_n$	$\in \{=_n, <_n\}$	comparisons on $\mathbb{Z}_n$
P	$:= x \sim_n y \mid x \sim_n l \mid x =_n y +_n z \mid x =_n a[i]$ $i \sim j \mid i \sim c \mid i = j + c \mid i = \lvert a\rvert$ $i \simeq x \mid a = b \mid a = b\{i \leftarrow x\}$	atomic formulas
F	$:= P \mid \neg F \mid F_1 \wedge F_2 \mid \exists v.F[v]$	$v \in Var$

Our array theory is *extensional*, that is, for $a, b \in \mathbb{A}$, we have $(a = b) \Leftrightarrow (\forall i.a[i] = b[i])$. Hence the notion of equality is extended to arrays.

3.1 Syntax

The syntax of UABE formulas is shown in Table 1. Note the comparison predicates on $\mathbb{Z}_n$ are different from those on $\mathbb{N}$ by the subscript n. Predicate $\simeq$ is defined on $\mathbb{N} \times \mathbb{Z}_n$, which establishes the equality between $\mathbb{N}$ and $\mathbb{Z}_n$.

In the index theory $T_{\mathbb{N}}$, addition of a variable and a constant, such as $i + 3$, is allowed, while addition of two variables, such as $i + j$, is forbidden. We will show later that addition of index variables leads to undecidability. In the element theory $T_{\mathbb{Z}_n}$, arbitrary arithmetic operators are allowed. Quantifiers can be freely used.

3.2 Semantics

A valuation[1] for UABE is a triple $V = (\mu, \nu, \tau)$, where

- $\mu : Var(\mathbb{Z}_n) \mapsto \mathbb{Z}_n$ assigns each element variable an integer in $\mathbb{Z}_n$,
- $\nu : Var(\mathbb{N}) \mapsto \mathbb{N}$ assigns each index variable a non-negative integer,
- $\tau : Var(\mathbb{A}) \mapsto \mathbb{Z}_n^*$ assigns each array variable a a *finite* sequence of $\mathbb{Z}_n$, i.e. $\tau(a) = a_0, a_1, \ldots, a_{|a|-1}$.

The *size* of array a is denoted by $|a|$. Array index should not exceed the array size. This condition is checked in all array reads. Although the size of an array is considered and checked here, it need not be given a concrete value. In other words, the size of an array is finite but can be arbitrarily large (we say it *unbounded*). For example, the proposition "for all integer m, there is an array a whose length is no less than m and all elements in a are identically 0" can be expressed in UABE as: $\forall m.\exists a.(|a| \geq m \wedge (\forall i < |a|.a[i] = 0))$.

Given a variable v, the value of v under valuation V is denoted as $V(v)$. Given a finite sequence $\tau(a)$, we denote $|\tau(a)|$ the length of it, $\tau(a)_p$ the p-th element of it, and $\tau(a)\{p \leftarrow v\}$ a new sequence by replacing the p-th element of $\tau(a)$ with v.

[1] For UABE we say *valuation* to distinguish from *interpretation* of WS1S.

Table 2. Semantics of UABE

(1)	$V \models (x \sim_n y)$	$\iff \mu(x) \sim \mu(y)$
(2)	$V \models (x \sim_n l)$	$\iff \mu(x) \sim l$
(3)	$V \models (x =_n y +_n z)$	$\iff \mu(x) = \mu(y) + \mu(z)$
(4)	$V \models (x =_n a[i])$	$\iff (\nu(i) < \lvert\tau(a)\rvert) \wedge (\mu(x) = \tau(a)_{\nu(i)})$
(5)	$V \models (i \sim j)$	$\iff \nu(i) \sim \nu(j)$
(6)	$V \models (i \sim c)$	$\iff \nu(i) \sim c$
(7)	$V \models (i = j + c)$	$\iff \nu(i) = \nu(j) + c$
(8)	$V \models (i = \lvert a\rvert)$	$\iff \nu(i) = \lvert\tau(a)\rvert$
(9)	$V \models (i \simeq x)$	$\iff \nu(i) = \mu(x)$
(10)	$V \models (a = b)$	$\iff \tau(a) = \tau(b)$
(11)	$V \models (a = b\{i \leftarrow x\})$	$\iff \tau(a) = \tau(b)\{\nu(i) \leftarrow \mu(x)\}$
(12)	$V \models \exists v.F[v]$	$\iff V \models F[v \leftarrow p], p$ is type consistent with v
(13)	$V \models \neg F$	$\iff V \not\models F$
(14)	$V \models F_1 \wedge F_2$	$\iff (V \models F_1) \wedge (V \models F_2)$

The semantics of UABE is listed in Table 2. Note that $\mathbb{Z}_n \subseteq \mathbb{N}$, hence the semantics of comparisons in (1) and (2) are comparisons between integer values. The semantics of addition of two variables in $\mathbb{Z}_n$ is defined in (3). Note that x, y, z are all n-bit integers, as a sanity condition we need to check the sum of $y +_n z$ does not exceed the range of n-bit integer. Since $\mu(x) < 2^n$ is already satisfied by the definition of μ, so is $\mu(y) + \mu(z) < 2^n$ (provided the equality holds). With rule (9), a variable in $\mathbb{N}$ and a variable in $\mathbb{Z}_n$ can be compared. Hence, the index theory and the element theory are related. As a result, the value of an element can be used as an index. In rule (10), $a = b$ means the sequences of $\tau(a)$ and $\tau(b)$ are identical, which implies $|a| = |b|$. The semantics of existential quantifier is defined in (12). We say p, v are type consistent, if $p \in \mathbb{N}$ and $v \in Var(\mathbb{N})$, or $p \in \mathbb{Z}_n$ and $v \in Var(\mathbb{Z}_n)$, or $p \in \mathbb{A}$ and $v \in Var(\mathbb{A})$.

3.3 Expressive Power

With the syntax in Table 1, many array formulas can be expressed. For example, any Boolean combinations of atomic formulas are expressible since $\{\neg, \wedge\}$ is sufficient to express $\{\vee, \rightarrow, \leftrightarrow\}$; universal quantifier is expressible since $\forall v.F[v] \iff \neg(\exists v.\neg F[v])$; minus function $-$ is expressible since it can be converted to $+$ by transposing the negative terms; comparison operations $\{\leq, >, \geq\}$ on index theory are expressible by using $\{\neg, <, +\}$; comparison operations $\{\leq_n, >_n, \geq_n\}$ on element theory are expressible by using $\{\neg, <_n, +_n\}$. Moreover, consecutive additions on elements, such as $x_1 +_n x_2 +_n x_3$, is also expressible, since one can replace $x_2 +_n x_3$ with a single variable y_2.

Although index and element terms are of different types, nested reads are allowed in UABE. When one want to use an $\mathbb{Z}_n$ term t as an index, one need to declare an existential-quantified variable t' in $Var(\mathbb{N})$, and then assert t and t' are equal. For example, the formula $z =_n a[a[i]]$ can be written in UABE as:

$\exists j.(z =_n a[j] \wedge j \simeq a[i])$. The property $\forall v \exists i.(b[v] > 0 \leftrightarrow a[i] = v)$ specified in the bucket sort algorithm can be expressed in UABE as

$$\forall v \exists i.\,((\exists v'.v' \simeq v \wedge 0 <_n b[v']) \leftrightarrow (v =_n a[i]))\,,$$

where $v' \in Var(\mathbb{N})$ is a fresh variable.

4 Decidability

In this section, we show the satisfiability and validity of formulas in UABE are decidable by giving a reduction to WS1S.

4.1 Representation of $\mathbb{Z}_n$ and Arrays in WS1S

Note a value in $\mathbb{Z}_n$ can be viewed as an n-bit integer. Given a term t of type $\mathbb{Z}_n$, t can be either a constant, a variable or an array read. We represent it as a bit vector $t = \overline{t^{[n-1]}t^{[n-2]}\cdots t^{[0]}}$, where $t^{[d]}$ is the d-th bit of t, $d = 0, 1, \ldots, n-1$, and $t^{[n-1]}$ is the most significant bit. If t is a constant or a variable, we encode the value of t as a single set $S_t = \{i | t^{[i]} = 1\}$ in WS1S. S_t is a subset of $\{0, 1, 2, \ldots, n-1\}$, and $d \in S_t$ iff $t^{[d]} = 1$. If t is an array read $a[i]$, since an array is actually a sequence of $\mathbb{Z}_n$ elements, we encode the array a as n finite sets $a^{(0)}, a^{(1)}, \ldots, a^{(n-1)}$, where $a^{(d)} = \{i | a[i]^{[d]} = 1\}$. Note here we do not encode each element of array as a single set, but the bit values of all elements in the same position as a single set. i.e $a[i]^{[d]} = 1$ iff $i \in a^{(d)}$.

For example, suppose $n = 4$, and all $\mathbb{Z}_n$ integers range over $\{0, 1, \ldots, 15\}$. Then the value $x = 5$ (in binary, $\overline{0101}$) can be encoded as $S_x = \{0, 2\}$, and the array a which is initialized with $a[0] = 1$ and $a[1] = 5$ can be encoded as $a^{(0)} = \{0, 1\}$, $a^{(1)} = \emptyset$, $a^{(2)} = \{1\}$ and $a^{(3)} = \emptyset$.

From above definitions, we can conclude that given any term t of type $\mathbb{Z}_n$, $t^{[d]}$ is a well-formed WS1S formula. Furthermore, the comparison and addition of $\mathbb{Z}_n$ values can be encoded in WS1S based on the bit-wise representations.

Lemma 1. *Equality relation $P_=(s,t)$ on $\mathbb{Z}_n$ can be encoded in WS1S.*

Proof. $P_=(s,t) \equiv \bigwedge_{d=0}^{n-1} \left(s^{[d]} \leftrightarrow t^{[d]}\right)$ □

Lemma 2. *Order relation $P_<(s,t)$ on $\mathbb{Z}_n$ can be encoded in WS1S.*

Proof. $P_<(s,t) \equiv \bigvee_{d=0}^{n-1} \left((\neg s^{[d]} \wedge t^{[d]}) \wedge \left(\bigwedge_{d'=d+1}^{n-1} (s^{[d']} \leftrightarrow t^{[d']})\right)\right)$ □

Lemma 3. *Addition on $\mathbb{Z}_n$ can be encoded in WS1S.*

Proof. Addition can be encoded as:

$$\begin{aligned} P_+(s,t,u) \equiv \exists C.(\ & C \subseteq \{1, \ldots, n-1\} \\ & \textstyle\bigwedge_{d=0}^{n-1}((d+1) \in C \leftrightarrow (d \in C \wedge t^{[d]} \vee d \in C \wedge u^{[d]} \vee t^{[d]} \wedge u^{[d]})) \\ & \textstyle\bigwedge_{d=0}^{n-1}(s^{[d]} \leftrightarrow \neg((d \in C) \leftrightarrow \neg(t^{[d]} \leftrightarrow u^{[d]})))) \end{aligned}$$

In the definition of $P_+(s,t,u)$, C is the add-carry. The first line restricts the sum not to exceed 2^n-1. The second line constraints $C^{[d+1]}=1$ whenever at least two of $t^{[d]}, u^{[d]}, C^{[d]}$ is 1. The third line constraints $s^{[d]} = (t^{[d]} \text{ xor } u^{[d]} \text{ xor } C^{[d]})$. It is guaranteed that $P_+(s,t,u) \iff s =_n t +_n u$. □

The domain of element theory can be generalized. If negative numbers are allowed in $\mathbb{Z}_n$, then a sign bit is needed. $P_=$, $P_<$ and P_+ can also be defined, although in a more complex form. Actually, a much richer set of operators can be supported. We can even replace the element domain $\mathbb{Z}_n$ with fixed size bit vectors. Bit-vector operations such as multiplication, division, modulo, etc, can be supported since they can also be encoded in WS1S.

4.2 Translation

A translation rule is denoted by a horizontal line which separates the original and translated formula. Above the line is the original formula in UABE, while the translated formula in WS1S is shown below the line.

In the translation, each index variable i in UABE is translated to a variable i in WS1S; each element variable x is translated to a set S_x and each array variable a corresponds to a variable $|a|$ and n sets: $a^{(0)}, \ldots, a^{(n-1)}$.

A special array $A_\simeq$ is used in the translation to establish the connection between $\mathbb{N}$ and $\mathbb{Z}_n$. We require $A_\simeq$ to be $(0,1,\ldots,2^n-1)$. Notice that $\mathbb{Z}_n \subseteq \mathbb{N}$. $A_\simeq$ is defined such that for indices $i < 2^n$, i and $A_\simeq[i]$ are equal in value. It is not a good idea to encode $A_\simeq$ one by one in WS1S. By observation, $A_\simeq^{(d)}$ is a set which does not contain $\{0,\ldots,2^d-1\}$ but contains $\{2^d,\ldots,2^{d+1}-1\}$. Furthermore, for numbers $i \geq 2^{d+1}$, $i \in A_\simeq^{(d)} \iff (i-2^{d+1}) \in A_\simeq^{(d)}$. For example, $A_\simeq^{(0)}$ contains all the 0-th (lowest) bit of array $(0,1,2,\ldots,2^n-1)$, i.e. $A_\simeq^{(0)} = \{1,3,5,7,\ldots\}$. Thus $A^{(d)}$ is defined directly in WS1S:

$$\begin{aligned} &\forall i.\left((i<2^d \rightarrow i \notin A_\simeq^{(d)}) \wedge (2^d \leq i \wedge i < 2^{d+1} \rightarrow i \in A_\simeq^{(d)})\right) \\ \bigwedge\ &\forall i.\left(i+2^{(d+1)} < 2^n \rightarrow ((i \in A_\simeq^{(d)}) \leftrightarrow (i+2^{(d+1)} \in A_\simeq^{(d)}))\right) \\ \bigwedge\ &\forall i.\left(i \geq 2^n \rightarrow i \notin A_\simeq^{(d)}\right) \end{aligned}$$

Use Θ to denote the conjunction of $|A_\simeq| = 2^n$ and all the definition of $A^{(d)}$ $(d = 0,1,\ldots,n-1)$. Thus, Θ is a WS1S formula which encodes $A_\simeq$. $A_\simeq$ is shared and reused during the translation.

Lemma 4. *Formula Θ is satisfiable.*

Proof. $A_\simeq = (0,1,\ldots,2^n-1)$, an interpretation which satisfies Θ can be obtained by encoding $A_\simeq$ into WS1S. □

Given a UABE formula, one can exhaustively apply the following translation rules until it is rewritten into WS1S. We will focus on the translation of atomic formulas, since both of UABE and WS1S do not have any restriction on quantifiers.

Table 3. Translation Rules for Atomic Formulas

$$(1.1)\frac{x =_n y}{P_=(x,y)} \quad (1.2)\frac{x <_n y}{P_<(x,y)} \quad (2.1)\frac{x =_n l}{P_=(x,l)} \quad (2.2)\frac{x <_n l}{P_<(x,l)} \quad (3)\frac{x =_n y +_n z}{P_+(x,y,z)}$$

$$(4)\frac{x =_n a[i]}{(i < |a|) \wedge P_=(x, a[i])} \quad (5)\frac{i \sim j}{i \sim j} \quad (6)\frac{i \sim c}{i \sim c} \quad (7)\frac{i = j + c}{i = j + c} \quad (8)\frac{i = |a|}{i = |a|}$$

$$(9)\frac{i \simeq x}{i < 2^n \wedge P_=(A_\simeq[i], x)} \quad (10)\frac{a = b}{(|a| = |b|) \wedge (\forall i.(i < |a|) \rightarrow P_=(a[i], b[i]))}$$

$$(11)\frac{a = b\{i \leftarrow x\}}{(i < |a|) \wedge P_=(a[i], x) \wedge |a| = |b| \wedge (\forall j.(j \neq i \wedge j < |a|) \rightarrow P_=(a[j], b[j]))}$$

Translation rules for atomic formulas are listed in Table 3. Each rule is tagged with a number. Rule $(x.y)$ in this table corresponds to item (x) in Table 2. Among these rules, (1.1), (1.2), (2.1), (2.2), (3), (4) are comparisons or addition on $\mathbb{Z}_n$, defined with the help of predefined shortcuts $P_=$, $P_<$ and P_+. In (4), we added the sanity condition for array read, $i < |a|$. (5), (6), (7), (8) are translated without any modification since they are already well-formed in WS1S. For (9), to translate atomic formula $i \simeq x$, the difficulty is type inconsistency between i and x: $i \in Var(\mathbb{N})$ but $x \in Var(\mathbb{Z}_n)$. With the help of $A_\simeq$, our method is replacing the original formula with $x =_n A_\simeq[i]$. In (10), $a = b$ implies the length of a, b are identical. In (11), array a and b are identical except at index i.

To handle quantifiers and Boolean combination of formulas, we add more rules, shown in Table 4. Denote $\Delta(\cdot)$ the translation procedure on atomic formulas which is defined in Table 3. In Table 4, (12.1), (12.2) and (12.3) handles quantifiers and (13), (14) handles Boolean combination of formulas. Boundary constraints Γ_1, Γ_2 are used. $\Gamma_1(x) \equiv S_x \subseteq \{0, 1, \ldots, n-1\}$, in (12.1) it is required because x should has at most n-bits. $\Gamma_2(a) \equiv \bigwedge_{d=0}^{n-1}(a^{(d)} \subseteq \{0, 1, \ldots, |a| - 1\})$, it is used in (12.3) to restrict the array size to $|a|$.

Table 4. Translation Rule for Complex Formulas

$$(12.1)\frac{\exists x.F}{\exists S_x.\Gamma_1(x) \wedge \Delta(F)} \quad (12.2)\frac{\exists i.F}{\exists i.\Delta(F)} \quad (12.3)\frac{\exists a.F}{\exists\,(|a|, a^{(0)}, \ldots, a^{(n-1)})\,.\Gamma_2(a) \wedge \Delta(F)}$$

$$(13)\frac{\neg F}{\neg\Delta(F)} \qquad (14)\frac{F_1 \wedge F_2}{\Delta(F_1) \wedge \Delta(F_2)}$$

Following the procedure above, these properties hold.

Theorem 1. *Given any UABE formula f and any valuation V, there is a corresponding interpretation σ_V for $\Delta(f)$, such that $\sigma_V \models \Theta$ and $V \models f \iff \sigma_V \models \Delta(f)$.*

Theorem 2. *Given any UABE formula f and any interpretation σ_V for $\Delta(f)$, if $\sigma_V \models \Theta$, then there is a valuation V for f such that $V \models f \iff \sigma_V \models \Delta(f)$.*

Theorem 1 and 2 actually show the fact that for each UABE formula f, there is a bijection between all valuations for f and all interpretations for $\Delta(f)$ that satisfies Θ. We give the idea how the connection between V and σ_V is established:

- $\forall i \in Var(\mathbb{N}).\ \sigma_V(i) = V(i)$,
- $\forall x \in Var(\mathbb{Z}_n).\forall d.(d \in \sigma_V(S_x) \iff V(x)^{[d]} = 1)$,
- $\forall a \in Var(\mathbb{A}).\ \sigma_V(|a|) = |V(a)|$,
- $\forall 0 \leq d < n.\forall a \in Var(\mathbb{A}).\forall i.(i \in \sigma_V(a^{(d)}) \iff V(a)^{[d]}_{V(i)} = 1$.

If $\sigma_V \models \Theta$, the above formulas already informally defined the bijection.

Theorem 3. *Given any UABE formula f, f is satisfiable if and only if $\Theta \wedge \Delta(f)$ is, f is valid if and only if $\Theta \rightarrow \Delta(f)$ is.*

Theorem 4. *The satisfiability for UABE is decidable.*

Proof. Our theory of UABE is decidable because WS1S is. □

Compared with other fragments of array theory, UABE has following characteristics: quantifiers can be freely used; nested reads of array are allowed; and there is no restriction on the structure of formula. Many array properties can be expressed in UABE, such as:

- Sorted: $\forall i, j.\ ((i < j < |a|) \rightarrow (a[i] \leq_n a[j]))$;
- Periodical: $\forall i.\ (i + T < |a| \rightarrow a[i] =_n a[i + T])$;
- Partitioned: elements with indices less than p are no larger than those with indices greater or equal to p:
  $\forall i, j.\ (i < p \leq j < |a| \rightarrow a[i] \leq_n a[j])$;
- Fibonacci array:
  $(a[0] =_n 1) \wedge (a[1] =_n 1) \wedge \forall i.\ (i + 2 < |a| \rightarrow a[i + 2] =_n a[i] +_n a[i + 1])$;
- an array monotonically increasing by 1 each step:
  $\forall i.(i < |a| \rightarrow a[i + 1] =_n a[i] +_n 1)$;

4.3 Infinite Arrays with Bounded Elements

In Sect. 3.2, the semantics of an array is referred as a *finite* sequence. We note that the semantics of an array can be extended to *infinite* sequence by a similar reduction to S1S without sacrificing decidability.

In the new theory, the semantics of arrays is changed to *infinite* sequence over $\mathbb{Z}_n$. $|a|$ is no longer needed because the size of an array can be infinite. In the translation, $a = b$ need not imply $|a| = |b|$ and the boundary constraint Γ_2 is no longer required. Then everything can work through. Nevertheless, in practice, if the target model is S1S, both finite and infinite interpretation of array can be mixed.

5 Extensions

In this section, we show that either extending the element theory to unbounded domain or allow addition of variables in the index theory such as $k = i + j$ will cause undecidability. In both cases, we show that the Hilbert's tenth problem [9] is reducible to the extended version of UABE. What we need to do is to define a formula $\varphi_+(k, i, j)$ which is satisfiable if and only if $k = i + j$, and a formula $\varphi_\times(k, i, j)$ which is satisfiable if and only if $k = i \times j$.

5.1 Extend Array Element to $\mathbb{N}$

We call this extended theory UAUE (Unbounded Array with Unbounded Elements). In this case, such $\varphi_+(k, i, j)$ and $\varphi_\times(k, i, j)$ can be defined in the same way as [4]. In particular, the $\forall^*\exists^*\forall^*\exists^*$-**SIL** logic fragment is then contained in UAUE. Undecidability of UAUE follows from [4].

5.2 Allow Addition on $\mathbb{N}$ Variables

We call this extended theory UABE$^+$. If we extend the theory of index to allow additions, $\varphi_+(k, i, j)$ can be defined as $k = i + j$. It remains to define $\varphi_\times(k, i, j)$. At first, assume i, j are both no less than 2.

The proof in [4] can not be applied because it relies on the fact that elements of an array can be arbitrarily large while the element theory is bounded in UABE$^+$. However, in UABE$^+$, we can assume the domain of array element contains at least two values, say $\{0, 1\}$. Then an array a can be treated as a finite set of integers which consists of all the indices where the corresponding array element is 1. Use $\hat{a}$ to denote the set represented by the array a. i.e: $\hat{a} = \{i \in \mathbb{N} | a[i] = 1\}$.

In this section, we also use:

- $[x, y]$ to denote the Least Common Multiple of x and y;
- $\langle x \rangle$ to denote the set $\{\alpha \cdot x | \alpha \in \mathbb{N}\}$, i.e, all multiple of x;
- $\langle x, y \rangle$ to denote the set $\{\alpha \cdot [x, y] | \alpha \in \mathbb{N}\}$, i.e, all multiple of $[x, y]$;

One can check immediately: $\forall x, y. \langle x, y \rangle = \langle [x, y] \rangle = \langle x \rangle \cap \langle y \rangle$

A finite set P is a *prefix* of some set Q (finite or infinite) if there is an integer r such that $P = \{0, 1, 2, \ldots, r\} \cap Q$. denoted by $P \sqsubset Q$. If P is a set, c is an integer, then $P + c = \{\alpha + c | \alpha \in P\}$ is the set obtained by adding c to each element in P.

The idea beyond the construction is expressed as a theorem:

Lemma 5. *Use $\Pi(i, j)$ to denote the set $= \langle i, j \rangle \cap (\langle i - 1, j - 1 \rangle + i + j - 1)$, if i, j are two positive integers and both are no less than 2, then the smallest number in $\Pi(i, j)$ is $i \times j$.*

Proof. Firstly, $ij - i - j + 1 = (i - 1)(j - 1)$, so $ij - i - j + 1 \in \langle i - 1, j - 1 \rangle$, Thus $ij \in (\langle i - 1, j - 1 \rangle + i + j - 1)$. Moreover, $ij \in \langle i, j \rangle$. Therefore $ij \in \Pi(i, j)$.

By contradiction, suppose there is an integer $p \in \Pi(i,j)$ and $p < ij$. Then $p \in \langle i,j\rangle$ and $p-i-j+1 \in \langle i-1, j-1\rangle$ both hold, which implies $i|p$, $j|p$, $(i-1)|(p-i-j+1)$ and $(j-1)|(p-i-j+1)$.

Suppose $[i,j] = ij/b$ where b is the Greatest Common Divisor of i and j. then $p = t[i,j] = t \cdot (ij/b)$ where $1 \le t < b$. Since $(i-1)|(p-i-j+1)$, we know:

$$\begin{aligned}
&(i-1)|(p-j) \Longleftrightarrow (i-1)|\left(\tfrac{tij}{b} - j\right)\\
\Longleftrightarrow\ &(i-1)|\left(((i-1)+1)\tfrac{tj}{b} - j\right) \Longleftrightarrow (i-1)|\left((i-1)\tfrac{tj}{b} + \tfrac{tj}{b} - j\right)\\
\Longleftrightarrow\ &(i-1)|\left(\tfrac{tj}{b} - j\right) \Longleftrightarrow (i-1)|(t-b)j/b
\end{aligned}$$

Symmetrically, $(j-1)|(t-b)i/b$. Hence $(i-1)(j-1)|(b-t)^2 \times j/b \times i/b$.

We assert that $(i-1) > (b-t) \times i/b$, it suffices to show $ti > b$: since $p \in \Pi(i,j)$, it is a necessary condition that $p \ge i+j-1$. i.e: $(tij/b \ge i+j-1) \Longleftrightarrow (tij/b > j) \Longleftrightarrow (ti > b)$. Symmetrically, we know $(j-1) > (b-t) \times j/b$. Hence $(i-1)(j-1) > (b-t)^2 \times j/b \times i/b$ holds. Combined with the fact $(i-1)(j-1)|(b-t)^2 \times j/b \times i/b$, we know $(b-t)^2 \times j/b \times i/b = 0$ must hold. But this is impossible because $t < b$ and $i, j \ge 2$. A contradiction. □

Following the idea presented in the previous lemma, we are willing to construct such $\varphi_\times(k,i,j)$. But the problem is, sets such as $\langle i,j\rangle$ are infinite. However, arrays should be finite in UABE$^+$. Good news is the finite prefixes of these infinite sets are sufficient to define $\varphi_\times(k,i,j)$, Because we can use a variable γ to bound the size of arrays, then quantify γ with an existential quantifier. For sufficiently large γ (larger than $i \times j$), everything works through. Detailed steps are:

- construct a set $\hat{M}_i \sqsubset \langle i\rangle$, by:

$$(M_i[0] =_n 1) \wedge (\forall p.(0 < p < i) \rightarrow M_i[p] =_n 0) \wedge (\forall p.p < \gamma \rightarrow M_i[p] =_n M_i[p+i])$$

- construct a set $\hat{M}_j \sqsubset \langle j\rangle$ in the same way;
- construct a set $\hat{M}_{i,j} \sqsubset \hat{M}_i \cap \hat{M}_j$, i.e: $\hat{M}_{i,j} \sqsubset \langle i,j\rangle$ by:

$$\forall p < \gamma.((M_{i,j}[p] =_n 1) \leftrightarrow (M_i[p] =_n 1 \wedge M_j[p] =_n 1))$$

- construct $\hat{M}_{i-1,j-1}$ in the same way;
- shift every element in $\hat{M}_{i-1,j-1}$ by $i+j-1$, result in $\hat{M}'_{i-1,j-1}$:

$$\begin{aligned}
&\forall p.((p < i+j-1) \rightarrow M'_{i-1,j-1}[p] =_n 0)\\
\wedge\ &\forall p < \gamma.M_{i-1,j-1}[p] =_n M'_{i-1,j-1}[p+i+j-1]
\end{aligned}$$

- find the minimal number in the intersection of $M_{i,j}$ and $M'_{i-1,j-1}$, say it k;

$$(M_{i,j}[k] = 1 \wedge M'_{i-1,j-1}[k] =_n 1) \wedge (\forall p.(M_{i,j}[p] =_n 1 \wedge M'_{i-1,j-1}[p] = 1) \rightarrow k \le p)$$

Let $\psi(k,i,j)$ be the conjunction of all the formulas where γ, M_i, M_j, $M_{i,j}$, M_{i-1}, M_{j-1}, $M_{i-1,j-1}$, $M'_{i-1,j-1}$ are existentially quantified. Because when γ is sufficiently large, such k will exists in the intersection of $M_{i,j}$ and $M'_{i-1,j-1}$. Then $\psi(k,i,j)$ is satisfiable if and only if $k = i \times j$.

When one of i, j is less than 2, trivial. Hence $\varphi_{\times}(k, i, j)$ can be defined as:

$$\begin{aligned}&((i = 0 \vee j = 0) \rightarrow k = 0)\\ \wedge\ &(i = 1 \rightarrow k = j) \wedge (j = 1 \rightarrow k = i)\\ \wedge\ &(i \geq 2 \wedge j \geq 2 \rightarrow \psi(k, i, j))\end{aligned}$$

Theorem 5. *The satisfiability for UABE$^+$ is undecidable.*

We mention the fact that "Presburger Arithmetic with Predicate is undecidable" is already proved in [5]. However it can not be used directly because array is finite, and finite arrays can not be used to represent predicates of Presburger arithmetic.

6 Example

As an illustrating example, we assume integers are 4-bit integers. Consider the formula F:

$$\exists a.(|a| = 10 \wedge \forall i.(i < |a| \rightarrow a[i + 1] =_n a[i] +_n 1))$$

This formula states: there exists an array a which increases by 1 each step, and the size of a is 10. Following the steps in Section 3.1, F is rewritten to UABE syntax, say F':

$$\begin{aligned}&\exists a.(|a| = 10 \wedge \neg\exists i.((i < |a|)\wedge\\ &\neg(\exists x, y, z, j.j = i + 1 \wedge z =_n 1 \wedge y =_n a[i] \wedge x =_n a[j] \wedge x =_n y +_n z)))\end{aligned}$$

Then F' is translated into $\Delta(F')$:

$$\begin{aligned}&\exists |a|, a^{(0)}, a^{(1)}, a^{(2)}, a^{(3)}.\Gamma_2(a) \wedge (P_{=}(|a|, 10) \wedge \neg\exists i.(P_{<}(i < |a|)\wedge\\ &\neg(\exists S_x, S_y, S_z, j.j = i + 1 \wedge P_{=}(z, 1) \wedge P_{=}(y, a[i]) \wedge P_{=}(x, a[j]) \wedge P_{+}(x, y, z))))\end{aligned}$$

MONA is used to check $\Theta \wedge \Delta(F')$. In a second, the result shows "satisfiable", and an example is given: $a = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$.

Many other formulas can be checked, such as:

- Fibonacci array is sorted: Because Fibonacci array increases exponentially. For a 4-bit element domain, we have to restrict $|a| < 7$ so that the value does not exceed $2^4 - 1 = 15$. The formula is given as:

$$\begin{aligned}&(|a| < 7) \wedge (a[0] =_n 1) \wedge (a[1] =_n 1)\\ &\wedge\forall i.i + 2 < |a| \rightarrow (a[i + 2] =_n a[i + 1] +_n a[i])\\ \rightarrow\ &(\forall i, j.i < j < |a| \rightarrow a[i] \leq_n a[j])\end{aligned}$$

  In a second, the result shows "valid", i.e. all Fibonacci array with length less than 7 is sorted.

– all values stored in array a are distinct:

$$\forall i, j.(i < |a| \wedge j < |a| \wedge (a[i] =_n a[j]) \rightarrow (j = i))$$

Check this formula, result is "satisfiable", and an example where all elements are distinct is given.

– other properties such as: if $a[i] < i$ for all indices then $a[a[i]] < i$ for all indices can be expressed as:

$$\forall i.(i < |a| \rightarrow a[i] < i) \rightarrow \forall i.(i < |a| \wedge a[i] < |a| \rightarrow a[a[i]] < i)$$

This formula is "valid".

As we can see, UABE has good expressive power. Many formulas that are not expressible in other logic fragments can be expressed and checked.

7 Conclusion

In this paper, we investigated a new array theory UABE. It is quite expressive but decidable. A decision procedure is given by translating UABE into WS1S. Two extensions of UABE are shown to be undecidable. Some examples of UABE formulas and their verification results are given. In the future, we want to implement the translation and do more experiments. We also want to see how it performs in program verification.

References

1. Bozga, M., Habermehl, P., Iosif, R., Konečný, F., Vojnar, T.: Automatic verification of integer array programs. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 157–172. Springer, Heidelberg (2009)
2. 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 (2005)
3. Büchi, J.: Weak second-order arithmetic and finite automata (1959)
4. Habermehl, P., Iosif, R., Vojnar, T.: A logic of singly indexed arrays. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. LNCS (LNAI), vol. 5330, pp. 558–573. Springer, Heidelberg (2008)
5. Halpern, J.Y.: Presburger arithmetic with unary predicates is Π_1^1 complete. Journal of Symbolic Logic 56, 56–62 (1991)
6. Henriksen, J.G., Jensen, O.J., Jørgensen, M.E., Klarlund, N., Paige, R., Rauhe, T., Sandholm, A.B.: 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)
7. Klarlund, N.: Mona & Fido: The logic-automaton connection in practice. In: Nielsen, M., Thomas, W. (eds.) CSL 1997. LNCS, vol. 1414, pp. 311–326. Springer, Heidelberg (1998)
8. Klarlund, N., Møller, A.: MONA Version 1.4 User Manual. BRICS, Department of Computer Science, Aarhus University, Notes Series NS-01-1 (2001), http://www.brics.dk/mona/ (revision of BRICS NS-98-3)

9. Matiyasevich, Y.: Enumerable sets are diophantine. Journal of Sovietic Mathematics, 354–358 (1970)
10. Mccarthy, J.: Towards a mathematical science of computation. In: IFIP Congress, pp. 21–28. North-Holland, Amsterdam (1962)
11. Möller, M., Rueß, H.: Solving bit-vector equations. In: Gopalakrishnan, G.C., Windley, P. (eds.) FMCAD 1998. LNCS, vol. 1522, p. 524. Springer, Heidelberg (1998)
12. Nelson, C.G.: Techniques for program verification. PhD thesis, Stanford University, Stanford, CA, USA (1980)
13. Stump, A., Barrett, C.W., Dill, D.L., Levitt, J.: A decision procedure for an extensional theory of arrays. In: LICS '01, Washington, DC, USA, pp. 29–37. IEEE Computer Society Press, Los Alamitos (2001)
14. Suzuki, N., Jefferson, D.: Verification decidability of presburger array programs. JACM 27(1), 191–205 (1980)

Quantifier Elimination by Lazy Model Enumeration*

David Monniaux

CNRS/VERIMAG**

Abstract. We propose a quantifier elimination scheme based on nested lazy model enumeration through SMT-solving, and projections. This scheme may be applied to any logic that fulfills certain conditions; we illustrate it for linear real arithmetic. The quantifier elimination problem for linear real arithmetic is doubly exponential in the worst case, and so is our method. We have implemented it and benchmarked it against other methods from the literature.

1 Introduction

Quantifier elimination consists in transforming a quantified formula into an equivalent quantifier-free formula. For instance, the formulas $\forall y\ (y - z \geq x \Rightarrow x + z \geq 1)$ and $x \geq 1 - z$ are equivalent (they have the same models for (x, z)), whether considered over the reals or integers. Quantifier elimination subsumes both satisfiability testing for quantifier-free formulas, and the decision of quantified formulas without free variables. In program analysis, quantifier elimination has been applied to obtain optimal invariants and optimal abstract transformers [1,2], and to obtain preconditions for modular assertion checking [3].

Unfortunately, quantifier elimination tends to be slow; as recalled in §4, worst-case complexities for useful theories tend to be towers of exponentials. Yet, high worst-case complexity does not preclude exploring procedures that perform fast on most examples, as shown by the high success of SAT solving. This motivates our work on new quantifier elimination algorithms.

Many interesting mathematical theories admit quantifier elimination. In order to introduce better elimination schemes, we shall first describe a naive, but inefficient algorithm (§2.2) which works by calling a *projection* operator, that is, an algorithm taking as an input a conjunction C of literals of the theory, and a list of variables $x_1, \ldots, x_n$, and outputting a formula equivalent to $\exists x_1, \ldots, x_n\ C$. Examples of theories with projection operators include nonlinear complex arithmetic (also known as the theory of algebraically closed fields), using Gröbner bases [4]; linear real arithmetic, using Fourier-Motzkin elimination [5, §5.4] or more advanced polyhedral projection techniques; and linear integer arithmetic, also known as Presburger arithmetic, using the Omega test [6].

* This work was partially funded by ANR project "ASOPT".

** VERIMAG is a joint laboratory of CNRS, Université Joseph Fourier and Grenoble-INP.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 585–599, 2010.

Nonlinear integer arithmetic, also known as Peano arithmetic, is undecidable. However, nonlinear (polynomial) real arithmetic[1] admits quantifier elimination. The best known general algorithms construct a *cylindrical algebraic decomposition* of the polynomials present in the atoms of the formula; once this costly decomposition is obtained, the quantifier elimination is trivial [7,8]. We therefore exclude these two theories from our study.

This article provides two contributions. First, it describe an algorithm that uses both projection and satisfiability testing modulo the chosen theory, and illustrate it with linear real arithmetic. This algorithm performs nested satisfiability tests, with lazy generation of constraints. Second, we improve on the worst case complexity bounds for an earlier algorithm [9], which are also valid for the new one.

In §2, we give a short introduction to quantifier elimination techniques over linear real arithmetic, and the idea of lazy constraint generation. In §3, we describe our algorithm, and we prove that its complexity is at most doubly exponential in §4. Finally, in §5 we provide benchmarks.

2 Previous State of the Art

Let us first recall some vocabulary on formulas. We shall then summarize previous work on quantifier elimination on linear real arithmetic, most notably our eager projection method (§2.2). We propose a lazy projection method using ideas of *lazy constraint generation*; §2.3 gives examples of such techniques in other applications.

2.1 Formulas

We consider quantifier-free formulas written using $\wedge$, $\vee$ and $\neg$ connectors, as well as literals (atoms or negation thereof). A formula written without $\neg$ except just around an atom is said to be in *negation normal form* (NNF), a formula consisting in a disjunction of conjunctions of literals is in *disjunctive normal form* (DNF), a formula consisting in a conjunction of disjunctions of literals is in *conjunctive normal form* (CNF).

We shall mostly focus on the case where the atoms are of the form $\sum_i a_i x_i \leq b$ where the a_i and b are constant rational numbers and the x_i are real variables. A model of a formula F is an assignment to the x_i such that the formula is satisfied; we then note $(x_1, \ldots, x_n) \vDash F$. We say that two formulas F and G are equivalent, noted $F \equiv G$, if they have the same models. We say that F implies G, noted $F \Rrightarrow G$, if all models of F are models of G. Formulas without free variables are equivalent to true or false. A *decision procedure* provides this truth value given such a formula. Obviously, a quantifier elimination procedure may be used as a decision procedure, since it will turn any formula into an equivalent formula without quantifiers or variables, thus trivially checkable.

[1] Also known as the theory of real closed fields.

We add to this language the $\forall$ and $\exists$ quantifiers. The definitions for models, equivalence and implication are the same as above, except that models only assign values to the free variables of the formula. Quantifier elimination consists in obtaining an equivalent quantifier-free formula from a quantified formula.

There exist two major classes of algorithms for quantifier elimination over arithmetic. One is based on substitution: an infinite disjunction $\exists x\ F$ is shown to be equivalent to a finite disjunction $F[x_1/x] \vee \cdots \vee F[x_n/x]$ where the x_i are functions of the free variables in F constructed by examination of the atoms in F. For linear real arithmetic, Ferrante and Rackoff's method [10] and Loos and Weispfenning's method [11] belong to this class, and so does Cooper's method for Presburger arithmetic [12]. Other methods, more geometrical in kind, project conjunctions of atoms and thus need some form of conversion to DNF; such is the case of Fourier-Mozkin elimination for linear real arithmetic, and of Pugh's Omega test for Presburger arithmetic [6]. Our methods belong to that latter class.

2.2 Eager Model Enumeration Algorithm

It is easy to see that if there is an algorithm π for eliminating quantifiers from formulas of the form $\exists x_1, \ldots, x_n\ F$ where C is a quantifier-free conjunction of literals, then there is an algorithm, albeit an inefficient one, for eliminating quantifiers from any formula.

We reduce ourselves to the case of eliminating the existential quantifier from $\exists x_1, \ldots, x_n\ F$ where F is quantifier-free. We handle an existentially quantified formula $\exists x_1, \ldots, x_n\ F$ as follows: convert F to DNF $F_1 \vee \cdots \vee F_m$; the formula is then equivalent to $(\exists x_1, \ldots, x_n\ F_1) \vee \cdots \vee (\exists x_1, \ldots, x_n\ F_m)$, and thus to $\pi(\exists x_1, \ldots, x_n\ F_1) \vee \cdots \vee \pi(\exists x_1, \ldots, x_n\ F_m)$. In the simplest form, π can be performed by the Fourier-Motzkin algorithm, and conversion to DNF by repeat application of the distributivity of $\wedge$ over $\vee$.

Can we do better? A more efficient way to convert to DNF is to use an "all-SAT" approach within a *satisfiability modulo theory* (SMT) solver. Given a quantifier-free formula F over linear real arithmetic, an SMT-solver will either answer "unsat", in which case F is unsatisfiable, or provide a model for F — that is, a valuation for all the free variables such that F is satisfied. Equivalently, an SMT-solver may provide truth values for all atoms in F such that all valuations of the free variables of F for which the atoms in F have these truth values are models; in other words, it provides a conjunction C of literals from the atoms of F such that C implies F.

In order to convert a formula F to DNF, we run an SMT-solver over it. If it answers "unsat", we are done. Otherwise, it provides a conjunction of literals C_1 such that $C_1 \Rightarrow F$. Run the SMT-solver over $F \wedge \neg C_1$. If it answers "unsat", we are done. Otherwise, it provides a conjunction of literals C_2 such that $C_2 \Rightarrow F$. Run the SMT-solver over $F \wedge \neg C_1 \wedge \neg C_2$, etc. This algorithm terminates, because there is a finite number of atoms and thus a finite number of conjunctions C_i that can be built out of them, and the same conjunction cannot occur twice. At the end $C_1 \vee C_2 \vee \ldots$ is a DNF form for F.

There is still room for improvement. Consider the formula defining the vertices of a n-dimensional hypercube $F \triangleq (x_1 = 0 \vee x_1 = 1) \wedge \cdots \wedge (x_n = 0 \vee x_n = 1)$ and compare with the result of the quantifier elimination $\exists x_2, \ldots, x_n\ F \equiv x_1 = 0 \vee x_1 = 1$. Certainly it seems excessive to enumerate the 2^{n-1} disjuncts of the DNF of F whereas the final result only has 2 disjuncts.

We therefore suggested another improvement [9]. Instead of adding $\neg C_i$ to the constraints of the system, we add $\neg\pi(C_i)$. With this method, the number of calls to the SMT-solver is not the size of the DNF of F, but the size of the DNF for the eliminated form of $\exists v_1, \ldots, v_m\ F$.

This algorithm has a weakness: when applied to nested quantifiers, for instance, $\exists x_1 \forall x_2 \exists x_3\ F$, it will compute a full DNF for $\exists x_3\ F$, then a full CNF for $\forall x_2 \exists x_3\ F$, prior to computing the DNF for the full formula, and it will do so even if most conjuncts or disjuncts are actually useless. Consider for instance the following example:

$$F \triangleq \exists x \forall y \exists z\ z \geq 0 \wedge (x \geq z \wedge y \geq z \vee y \leq 1 - z) \tag{1}$$

This formula was produced by adding an extra z to $(x \geq 0 \wedge y \geq 0) \vee y \leq 1$, which is equivalent to $x \geq 0 \vee y \leq 1$.

Let us see how the eager algorithm performs on F. First, $\exists z\ z \geq 0 (x \geq z \wedge y \geq z \vee x \geq z - 1 \wedge y \geq 1 - z)$ is turned to DNF: $F_2 \triangleq (x \geq 0 \wedge y \geq 0) \vee y \leq 1$ or, perhaps with some better algorithm, $x \geq 0 \vee y \leq 1$. Then, $\forall y\ F_2$ is turned to CNF, that is, $F_1 \triangleq x \geq 0$, and then $\exists x\ F_1$ is turned into true. Now consider that instead of F_2, we had taken $F_2' \triangleq x \geq 0$; clearly $F_2' \Rrightarrow F_2$. $\forall y\ F_2'$ is then $x \geq 0$. In short, instead of computing a full DNF for F_2' we could have simply computed one term of it. This motivates our lazy algorithm.

2.3 Lazy Constraint Generation in Other Contexts

In short, when looking for a model for variable x of $\forall y \exists z\ F$, each disjunct in the DNF of $\exists z\ F$ is an additional constraint over x, but we do not wish to generate the full list of these constraints because some of them may not be actually needed. The idea of our algorithm is to try to solve the already known constraints, find a tentative solution, and find if this solution violates some yet unknown constraint; if so, we add this constraint to the system and resume our search for a solution. Before describing a formal version of the algorithm, we wish to note that lazy constraint generation approaches are already used in other contexts, in order to better convey the intuition of the method.

In operational research, it is not uncommon for constrained optimization problems to be specified using a very large number of constraints, so large that explicitly taking them all into account at once would be impractical. New constraints are "discovered" when the proposed solution violates them. In linear programming, such technique is known as *delayed column generation*. As early as 1954, it was observed that it was possible to solve large instances of the traveling salesman problem by dynamically generating the inequalities that a solution should satisfy, the full set of inequalities being "astronomically" large [13].

Almost all SMT-solving systems proceed by Boolean relaxation: in order to decide whether a formula F is satisfiable, they first replace all non-propositional atoms by propositional variables, using a dictionary, then solve the resulting system using SAT. If the resulting propositional system is unsatisfiable, then so is the original problem. If it is satisfiable, it is possible that the Boolean solution is absurd with respect to the theory: for instance, if it assigns true to the propositional variables corresponding to the atoms $x > 1$, $y > 3$, and $x + y < 0$. If this is the case, an additional Boolean constraint is added to the problem, excluding this inconsistent assignment (or, for better efficiency, an inconsistent generalization of this assignment). In short, we generate on demand, or *lazily*, the theory of the atoms of F (the absurd conjunctions of atoms of F), because an eager approach would generate an exponential number of Boolean constraints [5, §11.2].

Some recent proposals for SMT-solving over linear real arithmetic [14,15] do not use Boolean relaxation. Instead, they try solving the formula directly for the real variables: for a problem over x, y and z, if they realize that after choosing $x = x_0$ and $y = y_0$, there is no suitable z (once x and y are chosen, the solution set for z can be computed as a intervals), they deduce a constraint on x and y that excludes (x_0, y_0). When solving for x, y, constraints on x may be obtained. This approach has similarities to what we would obtain by applying the ideas of §3 to a $\exists x_1 \exists x_2 \ldots \exists x_n\ F$ formula.

In quantified Boolean solving for formulas of the form $\forall b_1, \ldots, b_m \exists c_1, \ldots, c_n\ F$ (2QBF), some proposed approaches [16, Alg. I] use two successive layers of SAT solvers, with the inner solver solving for $b_1, \ldots, b_m, c_1, \ldots, c_n$, initially for formula F, and the outer solver for $c_1, \ldots, c_n$, initially for formula true, with new constraints being lazily generated and accumulated into the outer solver. The algorithm we present in §3 can be understood as a generalization of this algorithm to arbitrary theories and arbitrary quantification depths.

3 Lazy Model Enumeration Algorithm

We shall now describe our lazy algorithm, instantiated on linear real arithmetic (first, the generalization sub-algorithm, then the main algorithm), and prove its correctness. Then we shall briefly investigate possibilities of extension.

3.1 Generalization Algorithm

At some point during the course of the main algorithm, we shall generate a conjunction $C_1 \wedge \cdots \wedge C_n$ that implies a formula F, but for efficiency we would prefer a conjunction of fewer terms $C_{i_1} \wedge \cdots \wedge C_{i_n}$ that still implies F. In other words, we wish to generalize the conjunction $C_1 \wedge \cdots \wedge C_n$ under the condition that $C_1 \wedge \cdots \wedge C_n \Rrightarrow F$. Ideally, we wish the subset $\{C_{i_1}, \ldots, C_{i_n}\}$ to be minimal.

Our condition is equivalent to $C_1 \wedge \cdots \wedge C_n \wedge \neg F$ being a contradiction; thus, from a set of constraints $\{C_1, \ldots, C_n, \neg F\}$ we wish to extract a minimal contradictory set, or, in the terminology of operation research, a *irreducible infeasible*

Algorithm 1. GENERALIZE($C_0, test$): given a set S_0 and a monotonic Boolean function *test* such that $test(S_0)$ is true, return $S \subseteq S_0$ such that $test(S)$ is true and S is minimal.

Require: $(S_0, test)$
Require: *test* is a function that takes as input a set S of literals and answers true or false. It is required to be monotonic: if $S_1 \subseteq S_2$, and $test(S_1)$ is true, then so is $test(S_2)$.
Require: S_0 is a set of literals such that $test(S_0)$ is true.
  $S := S_0$
  while $S \neq \emptyset$ **do**
    Choose a literal c in S
    if $test(S \setminus \{c\})$ is true **then**
      $S := S \setminus \{c\}$
    end if
  end while
Ensure: $test(S)$ is true
Ensure: $S \subseteq S_0$

This description intentionally omits a precise criterion for the choice of c, since the algorithm is correct regardless.

subset. The simplest algorithm for doing so is the *deletion filter* [17,18]. A difference is that in operation research contexts, all constraints are inequalities, while in our case, formula F is in general complex, with disjunctions. In fact, we do not even want to explicitly write formula F — this is the difference with our earlier eager algorithm. Instead, we use a function *test* that takes as input a set $S \triangleq \{C_1, \ldots, C_n\}$ of literals, and answers "true" if and only if $\neg F \wedge C_1 \wedge \cdots \wedge C_n$ is unsatisfiable, thus leading to Alg. 1. This procedure merely relies on *test* being monotonic as a function from the sets of literals, ordered by inclusion, to the Booleans.

Procedure GENERALIZE can be replaced by a more clever, "divide-and-conquer" approach, as in the MIN function in [19] (or the equivalent QuickXPlain from [20]). While this procedure is theoretically better, making fewer calls to *test*, it performs worse in practice (§5). In our case, *test* is a complex procedure, possibly making use of multiple layers of quantifier elimination and SMT-solving, all of which use caches; thus, the cost of multiple calls does not depend solely on the number of calls but also on the relationships between the calls.

3.2 Main Algorithm

If B is a set $\{v_1, \ldots, v_n\}$ of variables, we denote by $\forall_B F$ the formula $\forall v_1 \ldots \forall v_n\, F$ (and respectively for $\exists_B F$). In all our algorithms and reasoning, the order of the bound variables inside these "block quantifiers" will not matter, thus the notation is justified. For technical reasons, we allow empty quantifier blocks ($\forall_\emptyset$ and $\exists_\emptyset$). We note $\neg^n F$ the formula F if n is even, $\neg F$ if n is odd. We note $\mathrm{FV}(F)$ the set of free variables of formula F.

We consider a formula F_0 in prenex form, with alternating quantifier blocks: $\forall_{B_0}\exists_{B_1}\forall_{B_2}\ldots\neg^n F_n$. Without loss of generality, we can suppose that the B_i have pairwise empty intersection. Any quantified formula can be converted to this form with $\forall i > 0\ B_i \neq \emptyset$, but possibly with $B_0 = \emptyset$. More precisely, we note, for $0 \leq i < n$, $F_i = \forall_{B_i}\neg F_{i+1}$.

π_i is a quantifier elimination procedure for conjunctions: from a conjunction C it returns another conjunction C' such that $C' \equiv \exists_{B_i} C$; for linear real arithmetic, Fourier-Motzkin elimination or more clever methods of polyhedral projection are suitable.[2] It is not necessary that C' be a conjunction for most of our algorithm, except for the GENERALIZE procedure (this restriction will be lifted in §3.5).

The main algorithm is the function Q-TEST(i, C). It tests whether $F_i \wedge C$ is satisfiable, and if it is, it proposes a conjunction C' of literals such that $\mathrm{FV}(C') \subseteq \mathrm{FV}(F_i)$, $C' \Rrightarrow F_i$, and $C \wedge C'$ is satisfiable. It is defined by induction over $n - i$ for $0 \leq i \leq n$.

The case $i = n$ corresponds to is merely SMT-solving and generalization. We note SMT-TEST(C, F) the SMT-solver function, which takes as inputs two formulas C and F and returns a couple (b, C'). b is a Boolean, which states whether $C \wedge F$ is satisfiable. If b is true, C' is an "extended model": a conjunction of literals of F such that $C' \Rrightarrow F$ and $C \wedge C'$ is satisfiable. Such a function can be obtained from an ordinary SMT-solver providing satisfiability models as follows: get a model $M \vDash C \wedge F$, then set C' as the conjunction of all the atoms a of F such that $M \vDash a$ and of the negation of all the atoms a of F such that $M \nvDash a$; alternatively, some SMT-solvers directly output such a conjunction.

The recursive case for $i < n$ is defined by calling the recursive case for $i + 1$. Let us begin by some intuition of the workings of the algorithm. Recall that $F_i \triangleq \forall_{B_i}\neg F_{i+1}$; thus, if we had a DNF $D_1 \vee \cdots \vee D_l$ of F_{i+1}, we could turn it immediately into a CNF of F_i: $(\neg\pi_i(D_1)) \wedge \cdots \wedge (\neg\pi_1(D_l))$. Our goal is to test whether $C \wedge F_i$ is satisfiable, which is equivalent to testing whether the set of constraints $\{C, \neg\pi_i(D_1), \ldots, \neg\pi_1(D_l)\}$ is satisfiable. Instead of computing all these constraints, then solving them, which is what our eager algorithm does, we wish to compute them "as needed".

The constraints that have already been computed at level i are stored as a current formula M_i (in practice, the current constraint state of an SMT-solver), initialized to true. Each of these formulas, for $0 < i < n$, satisfies two invariants: $\mathrm{FV}(M_i) \subseteq \mathrm{FV}(F_i)$ and $F_i \Rrightarrow M_i$. Intuitively, if the output of π_i is always a

[2] Fourier-Motzkin elimination directly obtains a set of inequalities defining the projected polyhedron, but it may create many unnecessary ones and it is thus often necessary to run tests for removing useless ones. Such tests are emptyness tests for polyhedra defined by constraints; these can be performed using the simplex algorithm implemented in exact rational arithmetic. Alternatively, methods based on the "double representation" of polyhedra first compute the set of vertices of the polyhedron (which may be exponential, for instance for a hypercube $[0, 1]^n$), project them (a trivial operation) then compute the facets of the resulting polyhedron. See [21] for a bibliography on polyhedral algorithms. Our implementation uses an off-the-shelf polyhedron library based on double representation; profiling has shown that the choice of the projection algorithm did not matter much [9].

Algorithm 2. Q-TEST(i, C): satisfiability testing for $F_i \wedge C$

Require: (i, C) such that $0 \leq i \leq n$, $\mathrm{FV}(C) \subseteq \mathrm{FV}(F_i)$
  if $i = n$ **then**
    $(b', C') :=$ SMT-TEST(C, F_n)
    if b' is false **then**
      return (false, false)
    else
      return (true, GENERALIZE$(C', K \mapsto \neg \mathit{first}(\text{SMT-TEST}(K, F_n))))$
    end if
  else
    while true do
      $(b', C') :=$ SMT-TEST(C, M_i)
      if b' is false **then**
        return (false, false) {Since $F_i \Rrightarrow M_i$, then $C \wedge F_i$ is unsatisfiable too}
      else
        $(b'', C'') :=$ Q-TEST$(i + 1, C')$
        if b'' is false **then**
          $\{C' \wedge F_{i+1}$ is unsatisfiable$\}$
          return (true, GENERALIZE$(C', K \mapsto \neg \mathit{first}(\text{Q-TEST}(i + 1, K))))$
        else
          $\{C''$ is such that $\mathrm{FV}(C'') \subseteq \mathrm{FV}(F_{i+1})$, $C'' \Rrightarrow F_{i+1}$ and $C' \wedge C''$ satisfiable.
          Thus $\exists_{B_i} C'' \Rrightarrow \exists_{B_i} F_{i+1}$, whence $F_i = \forall_{B_i} \neg F_{i+1} \Rrightarrow \neg \exists_{B_i} C'' \equiv \neg \pi_i(C'')\}$
          $M_i := M_i \wedge \neg \pi_i(C'')$
        end if
      end if
    end while
  end if
Ensure: The return value is a pair (b, C'). b is a Boolean stating whether $F_i \wedge C$ is satisfiable. If b is true, then C' is a conjunction of literals such that $\mathrm{FV}(C') \subseteq \mathrm{FV}(F_i)$, $C' \Rrightarrow F_i$, and $C \wedge C'$ is satisfiable.

$x \mapsto v$ denotes the function mapping x to v; *first* denotes the function mapping a couple to its first element.

conjunction, M_i is a "partial CNF" for F_i. At worst, the algorithm completes it into a full CNF for F_i.

The algorithm at level $i < n$ works as follows. It first tests satisfiability with respect to the already computed constraints: whether $C \wedge M_i$ is satisfiable; if it is not, then *a fortiori* $C \wedge F_i$ is not and the answer is immediate. Otherwise, we obtain an extended model C' of $C \wedge M_i$. We however do not know yet whether it is an extended model of $C \wedge F_i$; this is the case if and only if $C' \wedge F_{i+1}$ is unsatisfiable. We thus perform a recursive call to Q-TEST at level $i + 1$:

- If this call answers that $C' \wedge F_{i+1}$ is unsatisfiable, we could immediately return C' as a correct generalized model. Yet, C' might not be general enough: we would prefer to extract from it a minimal conjunction $C'_{\min}$ such that $C'_{\min} \wedge F_{i+1}$ is still unsatisfiable; thus the call to GENERALIZE. GENERALIZE has to test the satisfiability of various formulas of the form $C'_s \wedge F_{i+1}$; we

therefore supply it with the $K \mapsto \neg \mathit{first}(\text{Q-TEST}(i+1, K))$ function, which answers whether $K \wedge F_{i+1}$ is unsatisfiable.
- If $C' \wedge F_{i+1}$ is satisfiable, we obtain an extended model C'': $C'' \Rrightarrow F_{i+1}$. We therefore add $\neg\pi_i(C'')$ as a new constraint in M_i and retry solving.

3.3 Correctness

The correctness of GENERALIZE is obvious. The partial correctness of Q-TEST algorithm is proved by induction over $n-i$: we show it is correct for levels $i = n$ down to $i = 0$. For $i = n$, its correctness reduces to that of SMT-solving and GENERALIZE. The interesting case is $i < n$.

As noted in the algorithm description, we maintain the invariant $F_i \Rrightarrow M_i$. If $C \wedge M_i$ is unsatisfiable, then a fortiori $C \wedge F_i$ is unsatisfiable and the $(\mathsf{false}, \mathsf{false})$ answer is correct. Assume now the induction hypothesis: the correctness of Q-TEST$(i+1, C)$, which answers whether $C' \wedge F_{i+1}$ is unsatisfiable. If it is so, then $C' \Rrightarrow \forall_{B_i} \neg F_{i+1}$; we then generalize C' and answer the generalized version. Otherwise, we obtain $C'' \Rrightarrow F_{i+1}$; therefore $F_i \Rrightarrow \neg\pi_i(C'')$ and we can add $\neg\pi_i(C'')$ as a constraint in M_i.

Total correctness is ensured by the fact that the number of C' that can be generated at a given level i is finite, which is proved, again, by induction from $i = n-1$ down to $i = 0$. At level $n-1$, all the C' that we obtain are conjunctions of literals built from atoms of M_i. M_i is a conjunction of negations of projections of conjunctions of atoms of M_{i+1}. By the induction hypothesis, only a finite number of atoms can accumulate into M_{i+1}, thus only a finite number of constraints can accumulate into M_i, and the induction is proved. §4.2 provides complexity bounds.

3.4 Example

Recall the formula from Eq. 1: $\exists x \forall y \exists z\ z \geq 0 \wedge (x \geq z \wedge y \geq z \vee y \leq 1 - z)$. Its truth value is equivalent to the satisfiability of F_0: We therefore have

$$F_0 \triangleq \forall y \exists z\ z \geq 0 \wedge (x \geq z \wedge y \geq z \vee y \leq 1 - z) \tag{2}$$

$$F_1 \triangleq \forall z\ \neg(z \geq 0 \wedge (x \geq z \wedge y \geq z \vee y \leq 1 - z)) \tag{3}$$

$$F_2 \triangleq z \geq 0 \wedge (x \geq z \wedge y \geq z \vee y \leq 1 - z) \tag{4}$$

We initialize $M_0 = M_1 = \mathsf{true}$. Consider the call Q-TEST$(0, \mathsf{true})$. SMT-TEST (true, M_0) returns $(\mathsf{true}, \mathsf{true})$. Q-TEST$(1, \mathsf{true})$ is then called.

SMT-TEST(true, M_1) returns $(\mathsf{true}, \mathsf{true})$. Q-TEST$(2, \mathsf{true})$ is then called. This results in SMT-TEST(F_2, true) being called. SMT-solving of F_2 results in a "satisfiable" answer with a solution, for instance, $(x = 0, y = 0, z = 0)$; thus SMT-TEST(F_2, true) is $(\mathsf{true}, z \geq 0 \wedge x \geq z \wedge y \geq z \wedge y \leq 1 - z)$. GENERALIZE yields the simpler conjunction $z \geq 0 \wedge x \geq z \wedge y \geq z$, which still implies F_2. The projection of this conjunction parallel to z is $x \geq 0 \wedge y \geq 0$; we add its negation $x < 0 \vee y < 0$ to M_1. We again run SMT-TEST(true, M_1), which returns $(\mathsf{true}, x < 0)$. Q-TEST$(2, x < 0)$ is then called. This results in

SMT-TEST$(F_2, x < 0)$ being called. SMT-solving of $F_2 \wedge x < 0$ results in a "satisfiable" answer with a solution, for instance, $(x = -1, y = 0, z = 0)$; thus SMT-TEST$(F_2, x < 0)$ is $(\mathsf{true}, z \geq 0 \wedge y \leq 1 - z)$. GENERALIZE does not simplify this conjunction. The projection of this conjunction parallel to z is $y \leq 1$; we add its negation $y > 1$ to M_1, which is then $y > 1 \wedge (x < 0 \vee y < 0)$. We again run SMT-TEST$(\mathsf{true}, M_1)$, which returns $(\mathsf{true}, x < 0 \wedge y > 1)$. Q-TEST$(2, x < 0 \wedge y > 1)$ is then called. This results in SMT-TEST$(F_2, x < 0 \wedge y > 1)$ being called, with answer $(\mathsf{false}, \mathsf{false})$. We then know that $x < 0 \wedge y > 1 \Rrightarrow F_1$. Q-TEST$(1, \mathsf{true})$ then returns $(\mathsf{true}, x < 0 \wedge y > 1)$.

The projection of this conjunction parallel to y is $x < 0$; we add its negation $x \geq 0$ to M_0. SMT-TEST(true, M_0) returns $(\mathsf{true}, x \geq 0)$. Q-TEST$(1, x \geq 0)$ is then called. SMT-TEST$(x \geq 0, M_1)$ then returns $(\mathsf{false}, \mathsf{false})$. Q-TEST$(0, \mathsf{true})$ then returns $(\mathsf{true}, x \geq 0)$.

3.5 Generalizations

The above algorithm tests the satisfiability of a quantified formula and provides a generalized model if there is one. It can be turned into a quantifier elimination procedure by model enumeration: run Q-TEST$(0, \mathsf{true})$, obtain a model $C_1 \Rrightarrow F$, run Q-TEST$(0, \neg C_1)$, obtain a model $C_2 \Rrightarrow F$, run Q-TEST$(0, \neg V_1 \wedge \neg C_2)$ etc. until Q-TEST returns $(\mathsf{false}, \mathsf{false})$, then $C_1 \vee C_2$ is a DNF for F. This loop terminates for the same reason as Q-TEST: the number of C that can be generated is less than the 2^a where a is the number of possible atoms for M_0.

The algorithm can be generalized to any theory for which there are an SMT-solving algorithm and a projection operator. Obviously, propositional logic is suitable, though specialized QBF solvers are likely to be more efficient. Suitable theories include Presburger arithmetic: current SMT solvers implement integer arithmetic by relaxation to real numbers and branch'n'cut or Gomory cuts, and projection can be done using Omega [6].

One problem is that Omega outputs a disjunction: the results from the "dark shadow", plus a finite number of results from the "gray shadow". The simple generalization scheme in GENERALIZE is then unsuitable. Recall that this algorithm attempts generalizing a conjunction C by removing each conjunct and checking whether the resulting conjunction is still suitable (using the *test* oracle for suitability). Alternatively, one may see this method as replacing atoms by true inside the formula and checking whether the resulting formula is still suitable — which is a correct method for generalizing any formula in negation normal form.

For the sake of simplicity, we have required that the formula be in prenex form. It is possible to generalize the algorithm as follows: given formulas F and C, answer whether $C \wedge F$ is satisfiable and, if so, provide C' such that $C \wedge C'$ is satisfiable and $C' \Rrightarrow F$. Such an algorithm can be defined by induction over F: our Q-TEST algorithm implements the case where F contains no quantifier, or is of the form $\forall \neg F$. The case for $\exists F$ is just the case for F followed by projection. The case for $F_1 \vee F_2$ first tests F_1 then, if unsuccessful, F_2. The case for $(F_1 \wedge F_2, C)$ where F_1 has no quantifier (if necessary, eliminate them) reduces to that of $(F_2, F_1 \wedge C)$.

4 Complexity

We shall now prove that the algorithms of §2.2 and §3.2 are at most doubly exponential.

4.1 Number of Faces in Projected Polyhedra

A *polytope* in dimension d is the convex hull of a finite number of points of $\mathbb{R}^d$. We recall the usual definitions of faces [22, §2.1, p. 39]: a *vertex* is a 0-face, an *edge* a 1-face,a *facet* a $(d-1)$-face. If one considers conventional tridimensional geometry, then these definitions fit the usual ones for vertices, edges and sides respectively.

The number of k-faces of a polytope with v vertices in a d-dimensional space can be bounded [22, ch. 4, 5]:

Theorem 1 (McMullen). *The maximal number of k-faces for a polytope with v vertices in a d-dimensional space is obtained for cyclic polytopes and thus is $f_k(v,d)$.*

The number of k-faces in a cyclic polytope (a particular kind of polytope whose definition [22, p. 82] is unimportant for our purposes) can be explicitly computed [22, Prop. 19, p. 86]:

Proposition 1. *The number $f_k(v,d)$ of k-faces of a cyclic polytope with v vertices in a d-dimensional space ($k < d$) is given by:*

$$f_k(v,2n) = \sum_{j=1}^{n} \frac{v}{v-j}\binom{v-j}{j}\binom{j}{k+1-j} \tag{5}$$

$$f_k(v,2n+1) = \sum_{j=0}^{n} \frac{k+2}{v-j}\binom{v-j}{j+1}\binom{j+1}{k+1-j} \tag{6}$$

Observe that $f_k(v,2n)$, as a polynomial in v, has at most degree n, and that its coefficient of degree n, if $k+1 \geq n$, is $\binom{n}{k+1-n}\cdot\frac{1}{n!} = \frac{1}{(k+1-n)!(2n-k-1)!} \leq 1$. $f_k(v,2n+1)$, as a polynomial in v, also has at most degree n, and its coefficient of degree n, if $k+1 \geq n$, is $\frac{k+2}{(n+1)!}\binom{n+1}{k+1-n} = \frac{k+2}{(k-n+1)!(2n-k)!} \leq 3$. It follows that when $v \to \infty$, $f_k(v,d) = O\left(v^{\lfloor d/2 \rfloor}\right)$. Furthermore, if $k+1 < \lfloor d/2 \rfloor$, then $f_k(v,d)$ has leading monomial $v^{k+1}/(k+1)!$.

Note that a facet of the projection of a polytope P along p coordinates necessarily corresponds to a $(d-1-p)$-face of P (as an example, the edges of a polygon obtained by projecting a tridimensional polyhedron correspond to some of the edges of the original polyhedron). By polytope duality [22, §2.2, p. 61], they correspond to p-faces of the dual polytope of P, whose vertices are the facets of P. Therefore:

Lemma 1. *The number of facets of the projection of a polytope with f facets in $\mathbb{R}^d$ into $\mathbb{R}^{d-p}$, along p coordinates, is at most $f_p(f,d)$.*

The results above are valid for bounded polytopes, whereas our algorithms operate on unbounded polyhedra. By adding at most $2n$ constraints of the form $x_i \leq \pm C$ with C a large enough constant, we can obtain a bounded polytope P' out of an unbounded polyhedron P. The facets of the projection of P are found among the facets of the projection of P'. From the above results we deduce:

Lemma 2. *The number of facets of the projection of a polyhedron with f facets in $\mathbb{R}^d$ into $\mathbb{R}^{d-p}$, along p coordinates, is $O(f^{\lfloor d/2 \rfloor})$ as $f \to \infty$. Furthermore, if $p+1 < \lfloor d \rfloor 2$, then it is $O(f^{p+1})$.*

4.2 Application to our Algorithms

Consider a formula F in prenex form with n atoms and m variables, from which p are quantified. We can immediately exclude trivial atoms (those equivalent to true or false) and simplify the formula accordingly. In the remaining formula, each atom delimits a half-space. The number of distinct polyhedra that can be constructed from these half-space is at most 2^n. At all levels of our algorithms, we work with facets from projections of these polyhedra. Applying Lemma 2, the number of projections along p given coordinates of these facets is $O(2^{n(p+1)})$ as $n \to \infty$. The model enumeration algorithm, at a given level, can enumerate at most 2^a choices of truth values for $a = O(2^{n(m+1)})$ atoms. Each choice represents one run of SMT-solving, whose cost is $O(2^{a+m})$. To summarize, the overall costs are in $O\left(2^{2^{cnm}}\right)$ for some constant c, thus $O\left(2^{2^{c|F|^2}}\right)$ where $|F|$ is the size of the formula.

In comparison, the substitution-based elimination procedures have complexity $2^{2^{c|F|}}$ [10,11], and this is a lower bound for real quantifier elimination [23]. Also, any *nondeterministic* decision procedure for quantified real arithmetic has at least exponential complexity in the worst case [24]; so restricting ourselves to decision problems in lieu of quantifier elimination in general is not likely to help much.

However, when it comes to doubly exponential complexities, all that matters from practical purposes is practical complexity: an algorithm that performs well in practice is preferrable to an algorithm with better theoretical bounds, but that tends to reach its theoretical complexity. This is why we implemented the various methods and performed benchmarks, as seen in the next section.

5 Implementation and Benchmarks

We implemented the algorithms of Ferrante and Rackoff [10], Loos and Weispfenning [11], our eager algorithm [9], and our lazy algorithm for linear real arithmetic (§3.2) into our MJOLLNIR tool.[3]

[3] The current version of MJOLLNIR is available from http://www-verimag.imag.fr/~monniaux/download/, as well as the benchmarks formulas.

Table 1. Number of decision problems solved. Each block of 300 formulas $B_1, \ldots, B_6$ was randomly generated; we provide the number of formulas that at least one of the methods proved to be true or false. Maximal memory allowed was 1 GiB and maximal time 300 s.

	B_1	B_2	B_3	B_4	B_5	B_6
True formulas	115	80	220	285	219	230
False formulas	159	134	23	0	32	4
Ferrante-Rackoff (FR) solves	199	150	203	53	178	185
Loos-Weispfenning (LW) solves	250	220	244	80	247	249
Monniaux eager (M1) solves	241	128	218	260	231	112
Monniaux lazy (M2) solves	276	187	238	285	248	143
M2 solves but LW does not	32	9	26	209	28	1
LW solves but M2 does not	6	42	32	4	27	107
M2 solves but M1 does not	35	59	26	25	23	33
M1 solves but M2 does not	0	0	6	0	6	2
FR solves but LW does not	11	11	8	34	7	6
LW solves but FR does not	62	81	49	61	76	70

Since algorithmic costs are sensitive to the kind of formula output (CNF, DNF or unconstrained), we preferred to test these procedures only on decision problems — those without free variables, for which the output is true or false. We generated random benchmarks in blocks of 300, of various kinds:

1. B_1 consists in formulas with 10 variables, with sparse atoms, of the form $\forall v_9 \exists v_8 \forall v_7 \exists v_6 \forall v_5 \exists v_4 \forall v_3 \exists v_2 \forall v_1 \exists v_0\ F$.
2. B_2 consists in formulas with 12 variables, with sparse atoms, of the form $\forall v_{11} \exists v_{10} \forall v_9 \exists v_8 \forall v_7 \exists v_6 \forall v_5 \exists v_4 \forall v_3 \exists v_2 \forall v_1 \exists v_0\ F$.
3. B_3 consists in formulas with 12 variables, with sparse atoms, of the form $\forall v_{11}, v_{10}, v_9, v_8 \exists v_7, v_6, v_5, v_4 \forall v_3, v_2, v_1, v_0\ F$.
4. B_4 consists in formula of the same form as B_3 but with a more complex Boolean structure in F.
5. B_5 consists in formulas with 18 variables, with sparse atoms, of the form $\forall v_{17}, v_{16}, v_{15}, v_{14}, v_{13}, v_{12} \exists v_{11}, v_{10}, v_9, v_8, v_7, v_6 \forall v_5, v_4, v_3, v_2, v_1, v_0\ F$.
6. B_6 consists in formula of the same form as B_5 but with a more complex Boolean structure in F and denser atoms.

Results are provided in Tab.1. Generally speaking, our model enumeration algorithms fail due to timeout (set at 300 s) while the substitution methods fail to out-of-memory (maximal memory 1 GiB); also, the lazy model enumeration algorithm tends to perform better than the eager algorithm, and Loos and Weispfenning's algorithm better than Ferrante and Rackoff's. Comparing the substitution methods to the model enumeration algorithms is difficult: depending on how the benchmarks are generated, one class of algorithms may perform significantly better than the other.

On some $\forall\exists$ formulas produced by the minimization step of [2], the lazy procedure performs somewhat more slowly (10–40%) than the eager procedure. This

seems to indicate that on examples where it is actually necessary to enumerate all items of the eliminated form of the subformulas, it is faster to do it eagerly rather than do it lazily — which tends to apply to any comparison of eager and lazy approaches.

In the model enumeration algorithms, most of the time is spent in the SMT-solver, not in the polyhedral projection.

We investigated alternatives to the GENERALIZE function: the MIN function from [19], and variants of the order that GENERALIZE and MIN follow when considering atoms (randomly shuffled, atoms with the variables quantified at the innermost level first, same with outermost level). Surprisingly, MIN tended to perform worse.

6 Conclusion

We have described a quantifier elimination algorithm for linear real arithmetic that uses nested SMT-solver calls and polyhedral projection, in order to lazily enumerate models. This algorithm is different from those commonly applied for this problem, which are based on replacing existential quantification by a finite disjunction, substituting well-chosen witnesses for the value of the quantified variable. Both kinds of algorithms have doubly exponential complexity in the worst case, which is unavoidable for this problem.

For practical purposes, these two kinds of algorithms behave differently: substitution methods occasionally attempt to construct very large intermediate formulas and finish with out-of-memory, while model enumeration methods occasionally run into high computation times. We have experimented both kinds of methods on various classes of formulas, and, depending on the quantification and Boolean structures of the formulas, one method is favored over the other. There is therefore no clear winner.

Acknowledgments. We wish to thank Scott Cotton and Goran Frehse as well as the anonymous referees for helpful comments and suggestions.

References

1. Monniaux, D.: Optimal abstraction on real-valued programs. In: Nielson, H.R., Filé, G. (eds.) SAS 2007. LNCS, vol. 4634, pp. 104–120. Springer, Heidelberg (2007)
2. Monniaux, D.: Automatic modular abstractions for linear constraints. In: POPL (Principles of programming languages). ACM, New York (2009)
3. Moy, Y.: Sufficient preconditions for modular assertion checking. In: Logozzo, F., Peled, D., Zuck, L.D. (eds.) VMCAI 2008. LNCS, vol. 4905, pp. 188–202. Springer, Heidelberg (2008)
4. Cox, D., Little, J., O'Shea, D.: Ideals, Varieties, and Algorithms: An Introduction to Computational Algebraic Geometry and Commutative Algebra, 3rd edn. Springer, Heidelberg (2007)
5. Kroening, D., Strichman, O.: Decision procedures. Springer, Heidelberg (2008)

6. Pugh, W.: The Omega test: a fast and practical integer programming algorithm for dependence analysis. In: Supercomputing '91, pp. 4–13. ACM, New York (1991)
7. Collins, G.: Quantifier elimination for real closed fields by cylindric algebraic decomposition. In: Brakhage, H. (ed.) GI-Fachtagung 1975. LNCS, vol. 33, pp. 134–183. Springer, Heidelberg (1975)
8. Basu, S., Pollack, R., Roy, M.F.: Algorithms in real algebraic geometry, 2nd edn. Algorithms and computation in mathematics. Springer, Heidelberg (2006)
9. Monniaux, D.: A quantifier elimination algorithm for linear real arithmetic. In: Cervesato, I., Veith, H., Voronkov, A. (eds.) LPAR 2008. Monniaux, D, vol. 5330, pp. 243–257. Springer, Heidelberg (2008)
10. Ferrante, J., Rackoff, C.: A decision procedure for the first order theory of real addition with order. SIAM J. on Computing 4(1), 69–76 (1975)
11. Loos, R., Weispfenning, V.: Applying linear quantifier elimination. The Computer Journal, Special issue on computational quantifier elimination 36(5), 450–462 (1993)
12. Cooper, D.C.: Theorem proving in arithmetic without multiplication. In: Meltzer, B., Michie, D. (eds.) Machine Intelligence, vol. 7, pp. 91–100. Edinburgh University Press, Edinburgh (1972)
13. Dantzig, G.B., Fulkerson, D.R., Johnson, S.M.: Solution of a large-scale traveling-salesman problem. In: 50 Years of Integer Programming 1958–2008, pp. 7–28. Springer, Heidelberg (2009)
14. McMillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to richer logics. In: Bouajjani, A., Maler, O. (eds.) Computer Aided Verification. LNCS, vol. 5643, pp. 462–476. Springer, Heidelberg (2009)
15. Cotton, S.: On Some Problems in Satisfiability Solving. PhD thesis, Université Joseph Fourier, Grenoble, France (June 2009)
16. Ranjan, D.P., Tang, D., Malik, S.: A comparative study of 2QBF algorithms. In: Hoos, H., Mitchell, D.G. (eds.) SAT 2004. LNCS, vol. 3542, Springer, Heidelberg (2005)
17. Chinneck, J.W.: Finding a useful subset of constraints for analysis in an infeasible linear program. INFORMS J. on Computing 9(2) (1997)
18. Chinneck, J.W.: Feasibility and infeasibility in optimization. Springer, Heidelberg (2008)
19. Bradley, A.R., Manna, Z.: Property-directed incremental invariant generation. Formal Aspects of Computing 20(4–5), 379–405 (2008)
20. Junker, U.: QuickXplain: Conflict detection for arbitrary constraint propagation algorithms. In: IJCAI '01 workshop CONS-1 (2001)
21. Bagnara, R., Hill, P.M., Zaffanella, E.: The Parma Polyhedra Library, version 0.9
22. McMullen, P., Shepard, G.C.: Convex polytopes and the upper bound conjecture. London Mathematical Society lecture note series, vol. 3. Cambridge University Press, Cambridge (1971)
23. Davenport, J.H., Heintz, J.: Real quantifier elimination is doubly exponential. J. Symb. Comput. 5(1-2), 29–35 (1988)
24. Fischer, M.J., Rabin, M.O.: Super-exponential complexity of Presburger arithmetic. In: Karp, R. (ed.) Complexity of Computation. SIAM–AMS proceedings, vol. 7, pp. 27–42. American Mathematical Society, Providence (1974)

Bounded Underapproximations*

Pierre Ganty[1], Rupak Majumdar[2], and Benjamin Monmege[3]

[1] IMDEA Software, Spain
[2] UCLA, USA
[3] ENS Cachan, France

Abstract. We show a new and constructive proof of the following language-theoretic result: for every context-free language L, there is a *bounded* context-free language $L' \subseteq L$ which has the same Parikh (commutative) image as L. Bounded languages, introduced by Ginsburg and Spanier, are subsets of regular languages of the form $w_1^* w_2^* \cdots w_k^*$ for some $w_1, \ldots, w_k \in \Sigma^*$. In particular bounded context-free languages have nice structural and decidability properties. Our proof proceeds in two parts. First, using Newton's iterations on the language semiring, we construct a context-free subset L_N of L that can be represented as a sequence of substitutions on a linear language and has the same Parikh image as L. Second, we inductively construct a Parikh-equivalent bounded context-free subset of L_N.

As an application of this result in model checking, we show how to underapproximate the reachable state space of multithreaded procedural programs. The bounded language constructed above provides a decidable underapproximation for the original problem. By iterating the construction, we get a semi-algorithm for the original problems that constructs a sequence of underapproximations such that no two underapproximations of the sequence can be compared. This provides a progress guarantee: every word $w \in L$ is in some underapproximation of the sequence, and hence, a program bug is guaranteed to be found. In particular, we show that verification with bounded languages generalizes context-bounded reachability for multithreaded programs.

1 Introduction

Many problems in program analysis reduce to undecidable problems about context-free languages. For example, checking safety properties of multithreaded recursive programs reduces to checking emptiness of the intersection of context-free languages [16,2].

We study underapproximations of these problems, with the intent of building tools to find bugs in systems. In particular, we study underapproximations in which one or more context-free languages arising in the analysis are replaced by

* This research was sponsored in part by the NSF grants CCF-0546170 and CCF-0702743, DARPA grant HR0011-09-1-0037, EU People/COFUND program 229599 AMAROUT, EU IST FET Project 231620 HATS, and Madrid Regional Government project S2009TIC-1465 PROMETIDOS.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 600–614, 2010.

their subsets in a way that (P1) the resulting problem after the replacement becomes decidable and (P2) the subset preserves "many" strings from the original language. Condition (P1) ensures that we have an algorithmic check for the underapproximation. Condition (P2) ensures that we are likely to retain behaviors that would cause a bug in the original analysis.

We show in this paper an underapproximation scheme using *bounded languages* [9,8]. A language L is *bounded* if there exist $k \in \mathbb{N}$ and finite words $w_1, w_2, \ldots, w_k$ such that L is a subset of the regular language $w_1^* \cdots w_k^*$. In particular, context-free bounded languages (hereunder bounded languages for short) have stronger properties than general context-free languages: for example, it is decidable to check if the intersection of a context-free language and a bounded language is non-empty [9]. For our application to verification, these decidability results ensure condition (P1) above.

The key to condition (P2) is the following *Parikh-boundedness* property: for every context-free language L, there is a bounded language $L' \subseteq L$ such that the Parikh images of L and L' coincide. (The *Parikh image* of a word w maps each symbol of the alphabet to the number of times it appears in w, the Parikh image of a language is the set of Parikh images of all words in the language.) A language L' meeting the above conditions is called a *Parikh-equivalent bounded subset* of L. Intuitively, L' preserves "many" behaviors as for every string in L, there is a permutation of its symbols that matches a string in L'.

The Parikh-boundedness property was first proved in [13,1], however, the chain of reasoning used in these papers made it difficult to see how to explicitly construct the Parikh-equivalent bounded subset. Our paper gives a direct and constructive proof of the theorem. We identify two contributions in this paper.

Explicit construction of Parikh-equivalent bounded subsets. Our constructive proof has two parts. First, using Newton's iteration [5] on the semiring of languages, we construct, for a given context-free language L, a finite sequence of linear substitutions which denotes a Parikh-equivalent (but not necessarily bounded) subset of L. (A linear substitution maps a symbol to a language defined by a *linear* grammar, that is, a context-free grammar where each rule has at most one non-terminal on the right-hand side.) The Parikh equivalence follows from a convergence property of Newton's iteration on the related commutative semiring.

Second, we provide a direct constructive proof that takes as input such a sequence of linear substitutions, and constructs by induction a Parikh-equivalent bounded subset of the language denoted by the sequence.

Reachability analysis of multithreaded programs with procedures. Using the above construction, we obtain a semi-algorithm for reachability analysis of multithreaded programs with the intent of finding bugs. To check if configuration (c_1, c_2) of a recursive 2-threaded program is reachable, we construct the context-free languages $L_1^0 = L(c_1)$ and $L_2^0 = L(c_2)$ respectively given by the execution paths whose last configurations are c_1 and c_2, and check if either $L_1' \cap L_2^0$ or $L_1^0 \cap L_2'$ is non-empty, where $L_1' = L_1^0 \cap w_1^* \cdots w_k^*$ and $L_2' = L_2^0 \cap v_1^* \cdots v_l^*$

are two Parikh-equivalent bounded subsets of L_1^0 and L_2^0, respectively. If either intersection is non-empty, we have found a witness trace. Otherwise, we construct $L_1^1 = L_1^0 \cap \overline{w_1^* \cdots w_k^*}$ and $L_2^1 = L_2^0 \cap \overline{v_1^* \cdots v_l^*}$ in order to exclude, from the subsequent analyses, the execution paths we already inspected. We continue by rerunning the above analysis on L_1^1 and L_2^1. If (c_1, c_2) is reachable, the iteration is guaranteed to terminate; if not, it could potentially run forever. Moreover, we show our technique subsumes and generalizes context-bounded reachability [15].

We omit proofs for space reasons. Detailed proofs, as well as one more application of our result, can be found in [7].

Related Work. Bounded languages were introduced and studied by Ginsburg and Spanier [9] (see also [8]). The existence of a bounded, Parikh-equivalent subset for a context-free language was shown in [1] using previous results on languages in the Greibach hierarchy [13]. The existence of a language representable as a sequence of linear transformations of a linear language which is Parikh-equivalent to a context-free language was independently shown in [6].

Bounded languages have been recently proposed by Kahlon for tractable reachability analysis of multithreaded programs [11]. His observation is that in many practical instances of multithreaded reachability, the languages are actually bounded. If this is true, his algorithm checks the emptiness of the intersection (using the algorithm in [9]). In contrast, our results are applicable even if the boundedness property does not hold.

For multithreaded reachability, *context-bounded reachability* [15,17] is a popular underapproximation technique which tackles the undecidability by limiting the search to those runs where the active thread changes at most k times. Our algorithm using bounded languages *subsumes* context-bounded reachability, and can capture unboundedly many synchronizations in one analysis. We leave the empirical evaluation of our algorithms for future work.

2 Preliminaries

We assume the reader is familiar with the basics of language theory (see [10]). An alphabet Σ is a finite non-empty set of symbols. The concatenation $L \cdot L'$ of two languages $L, L' \subseteq \Sigma^*$ is defined using word concatenation as $L \cdot L' = \{l \cdot l' \mid l \in L \wedge l' \in L'\}$.

An *elementary bounded language* over Σ is a language of the form $w_1^* \cdots w_k^*$ for some fixed $w_1, \ldots, w_k \in \Sigma^*$.

Vectors. For $p \in \mathbb{N}$, we write $\mathbb{Z}^p$ and $\mathbb{N}^p$ for the set of p-dim vectors (or simply vectors) of integers and naturals, respectively. We write $\mathbf{0}$ for the vector $(0, \ldots, 0)$ and $\mathbf{e}_i$ the vector $(z_1, \ldots, z_p) \in \mathbb{N}^p$ such that $z_j = 1$ if $j = i$ and $z_j = 0$ otherwise. *Addition* on p-dim vectors is the componentwise extension of its scalar counterpart, that is, given $(x_1, \ldots, x_p), (y_1, \ldots, y_p) \in \mathbb{Z}^p$ $(x_1, \ldots, x_p) + (y_1, \ldots, y_p) = (x_1 + y_1, \ldots, x_p + y_p)$. Using vector addition, we

define the operation $\dotplus$ on sets of vectors as follows: given $Z, Z' \subseteq \mathbb{N}^p$, let $Z \dotplus Z' = \{z + z' \mid z \in Z \wedge z' \in Z'\}$.

Parikh Image. Give Σ a fixed linear order: $\Sigma = \{a_1, \ldots, a_p\}$. The Parikh image of a symbol $a_i \in \Sigma$, written $\Pi_\Sigma(a_i)$, is $\mathbf{e}_i$. The Parikh image is extended to words of Σ^* as follows: $\Pi_\Sigma(\varepsilon) = \mathbf{0}$ and $\Pi_\Sigma(u \cdot v) = \Pi_\Sigma(u) + \Pi_\Sigma(v)$. Finally, the Parikh image of a language on Σ^* is the set of Parikh images of its words. Thus, the Parikh image maps 2^{Σ^*} to $2^{\mathbb{N}^p}$. We also define the inverse of the Parikh image $\Pi_\Sigma^{-1} : 2^{\mathbb{N}^p} \to 2^{\Sigma^*}$ as follows: given a subset M of $\mathbb{N}^p$, $\Pi_\Sigma^{-1}(M)$ is the set $\{y \in \Sigma^* \mid \exists m \in M : m = \Pi_\Sigma(y)\}$. When it is clear from the context we generally omit the subscript in Π_Σ and Π_Σ^{-1}.

Context-free Languages. A *context-free grammar* G is a tuple $(\mathcal{X}, \Sigma, \delta)$ where $\mathcal{X}$ is a finite non-empty set of variables (non-terminal letters), Σ is an alphabet of terminal letters and $\delta \subseteq \mathcal{X} \times (\Sigma \cup \mathcal{X})^*$ a finite set of productions (the production (X, w) may also be noted $X \to w$). Given two strings $u, v \in (\Sigma \cup \mathcal{X})^*$ we define the relation $u \Rightarrow v$, if there exists a production $(X, w) \in \delta$ and some words $y, z \in (\Sigma \cup \mathcal{X})^*$ such that $u = yXz$ and $v = ywz$. We use $\Rightarrow^*$ for the reflexive transitive closure of $\Rightarrow$. A word $w \in \Sigma^*$ is recognized by the grammar G from the state $X \in \mathcal{X}$ if $X \Rightarrow^* w$. Given $X \in \mathcal{X}$, the language $L_X(G)$ is given by $\{w \in \Sigma^* \mid X \Rightarrow^* w\}$. A language L is *context-free* (written CFL) if there exists a context-free grammar $G = (\mathcal{X}, \Sigma, \delta)$ and an initial variable $X \in \mathcal{X}$ such that is $L = L_X(G)$. A *linear grammar* G is a context-free grammar where each production is in $\mathcal{X} \times \Sigma^*(\mathcal{X} \cup \{\varepsilon\})\Sigma^*$. A language L is *linear* if $L = L_X(G)$ for some linear grammar G and initial variable X of G. A CFL L is *bounded* if it is a subset of some elementary bounded language.

Proof Plan. The main result of the paper is the following.

Theorem 1. *For every CFL L, there is an effectively computable CFL L' such that (i) $L' \subseteq L$, (ii) $\Pi(L) = \Pi(L')$, and (iii) L' is bounded.*

We actually solve the following related problem in our proof.

Problem 1. Given a CFL L, compute an elementary bounded language B such that $\Pi(L \cap B) = \Pi(L)$.

If we can compute such a B, then we can compute the CFL $L' = B \cap L$ which satisfies conditions (i) to (iii) of the Th. 1. Thus, solving Pb. 1 proves the theorem constructively.

We solve Pb. 1 for a language L as follows: (1) we find an L' such that $L' \subseteq L$, $\Pi(L') = \Pi(L)$, and L' has a "simple" structure (Sect. 3) and (2) then show how to find an elementary bounded B with $\Pi(L' \cap B) = \Pi(L')$, assuming this structure (Sect. 4). Observe that if $L' \subseteq L$ and $\Pi(L) = \Pi(L')$, then for every elementary bounded B, we have $\Pi(L' \cap B) = \Pi(L')$ implies $\Pi(L \cap B) = \Pi(L)$ as well. So the solution B for L' in step (2) is a solution for L as well. Section 5 provides an application of the result for multithreaded program analysis and compares it with an existing technique.

3 A Parikh-Equivalent Representation

Our proof to compute the above L' relies on a fixpoint characterization of CFLs and their Parikh image. Accordingly, we introduce the necessary mathematical notions to define and study properties of those fixpoints.

Semiring. A *semiring* $\mathcal{S}$ is a tuple $\langle S, \oplus, \odot, \bar{0}, \bar{1}\rangle$, where S is a set with $\bar{0}, \bar{1} \in S$, $\langle S, \oplus, \bar{0}\rangle$ is a commutative monoid with neutral element $\bar{0}$, $\langle S, \odot, \bar{1}\rangle$ is a monoid with neutral element $\bar{1}$, $\bar{0}$ is an annihilator w.r.t. $\odot$, *i.e.* $\bar{0} \odot a = a \odot \bar{0} = \bar{0}$ for all $a \in S$, and $\odot$ distributes over $\oplus$, *i.e.* $a \odot (b \oplus c) = (a \odot b) \oplus (a \odot c)$, and $(a \oplus b) \odot c = (a \odot c) \oplus (b \odot c)$. We call $\oplus$ the *combine* operation and $\odot$ the *extend* operation. The natural order relation $\sqsubseteq$ on a semiring $\mathcal{S}$ is defined by $a \sqsubseteq b \Leftrightarrow \exists d \in S\colon a \oplus d = b$. The semiring $\mathcal{S}$ is *naturally ordered* if $\sqsubseteq$ is a partial order on S. The semiring $\mathcal{S}$ is *commutative* if $a \odot b = b \odot a$ for all $a, b \in S$, *idempotent* if $a \oplus a = a$ for all $a \in S$, *complete* if it is naturally ordered and $\sqsubseteq$ is such that ω-chains $a_0 \sqsubseteq a_1 \sqsubseteq \cdots \sqsubseteq a_n \sqsubseteq \cdots$ have least upper bounds. Finally, the semiring $\mathcal{S}$ is *ω-continuous* if it is naturally ordered, complete and for all sequences $(a_i)_{i\in\mathbb{N}}$ with $a_i \in S$, $\sup\{\bigoplus_{i=0}^{n} a_i \mid n \in \mathbb{N}\} = \bigoplus_{i\in\mathbb{N}} a_i$. We define two semirings we shall use subsequently.

Language Semiring. Let $\mathcal{L} = \left\langle 2^{\Sigma^*}, \cup, \cdot, \emptyset, \{\varepsilon\}\right\rangle$ denote the idempotent ω-continuous semiring of languages. The natural order on $\mathcal{L}$ is given by set inclusion (viz. $\subseteq$).

Parikh Semiring. The tuple $\mathcal{P} = \left\langle 2^{\mathbb{N}^p}, \cup, \dot{+}, \emptyset, \{\mathbf{0}\}\right\rangle$ is the idempotent ω-continuous commutative semiring of Parikh vectors. The natural order is again given by $\subseteq$.

Valuation, polynomial. In what follows, let $\mathcal{X}$ be a finite set of variables and $\mathcal{S} = \langle S, \oplus, \odot, \bar{0}, \bar{1}\rangle$ be an ω-continuous semiring.

A *valuation* $\boldsymbol{v}$ is a mapping $\mathcal{X} \to S$. We denote by $\mathcal{S}^{\mathcal{X}}$ the set of all valuations and by $\ddot{0}$ the valuation which maps each variable to $\bar{0}$. We define $\dot{\sqsubseteq} \subseteq \mathcal{S}^{\mathcal{X}} \times \mathcal{S}^{\mathcal{X}}$ as the order given by $\boldsymbol{v} \dot{\sqsubseteq} \boldsymbol{v}'$ if and only if $\boldsymbol{v}(X) \sqsubseteq \boldsymbol{v}'(X)$ for every $X \in \mathcal{X}$. A *monomial* is a mapping $\mathcal{S}^{\mathcal{X}} \to S$ given by a finite expression $m = a_1 \odot X_1 \odot a_2 \cdots a_k \odot X_k \odot a_{k+1}$ where $k \geq 0$, $a_1, \ldots, a_{k+1} \in S$ and $X_1, \ldots, X_k \in \mathcal{X}$ such that $m(\boldsymbol{v}) = a_1 \odot \boldsymbol{v}(X_1) \odot a_2 \cdots a_k \odot \boldsymbol{v}(X_k) \odot a_{k+1}$ for $\boldsymbol{v} \in \mathcal{S}^{\mathcal{X}}$.

A *polynomial* is a finite combination of monomials: $f = m_1 \oplus \cdots \oplus m_k$ where $k \geq 0$ and $m_1, \ldots, m_k$ are monomials. The set of polynomials w.r.t. $\mathcal{S}$ and $\mathcal{X}$ will be denoted by $\mathcal{S}[\mathcal{X}]$. Finally, a *polynomial transformation* $\boldsymbol{F}$ is a mapping $\mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$ described by the set $\{\boldsymbol{F}_X \in \mathcal{S}[\mathcal{X}] \mid X \in \mathcal{X}\}$ of polynomials: hence, for every vector $\boldsymbol{v} \in \mathcal{S}^{\mathcal{X}}$, $\boldsymbol{F}(\boldsymbol{v})$ is a valuation of each variable $X \in \mathcal{X}$ to $\boldsymbol{F}_X(\boldsymbol{v})$.

Least Fixpoint. Recall that a mapping $f\colon S \to S$ is *monotone* if $a \sqsubseteq b$ implies $f(a) \sqsubseteq f(b)$, and *continuous* if for any infinite chain $a_0, a_1, a_2, \ldots$ we have $\sup\{f(a_i)\} = f(\sup\{a_i\})$. The definition can be extended to mappings $\boldsymbol{F}\colon \mathcal{S}^{\mathcal{X}} \to \mathcal{S}^{\mathcal{X}}$ in the obvious way (using $\dot{\sqsubseteq}$). Then we may formulate the following proposition (cf. [12]).

Proposition 1. *Let $\boldsymbol{F}$ be a polynomial transformation. The mapping induced by $\boldsymbol{F}$ is monotone and continuous and $\boldsymbol{F}$ has a unique least fixpoint $\mu\boldsymbol{F}$.*

Fixpoints of polynomial transformations relates to CFLs as follows. Given a grammar $G = (\mathcal{X}, \Sigma, \delta)$, let $L(G)$ be the valuation which maps each variable $X \in \mathcal{X}$ to the language $L_X(G)$. We first characterize the valuation $L(G)$ as the least fixpoint of a polynomial transformation $\boldsymbol{F}$ defined as follows: each $\boldsymbol{F}_X$ of $\boldsymbol{F}$ is given by the combination of α's for $(X, \alpha) \in \delta$ where α is interpreted as a monomial on the semiring $\mathcal{L}$. From [3] we know that $L(G) = \mu\boldsymbol{F}$.

Example 1. Let $G = (\{X_0, X_1\}, \{a, b\}, \delta)$ where $\delta = \{(X_0 \to aX_1|a), (X_1 \to X_0b|aX_1bX_0)\}$. It defines the polynomial transformation $\boldsymbol{F}$ on $\mathcal{L}^{\mathcal{X}}$ such that $\boldsymbol{F}_{X_0} = a \cdot X_1 \cup a$ and $\boldsymbol{F}_{X_1} = X_0 \cdot b \cup a \cdot X_1 \cdot b \cdot X_0$, and $L(G)$ is the least fixpoint of $\boldsymbol{F}$ in the language semiring. □

3.1 Relating the Language and Parikh Semirings

Given a polynomial transformation $\boldsymbol{F}$, we now characterize the relationship between the least fixpoints $\mu\boldsymbol{F}$ taken over the language and the Parikh semiring, respectively. Either fixpoint is given by the limit of a sequence of *iterates* which is defined by Newton's iteration scheme [4,5]. Our characterization operates at the level of those iterates: we inductively relate the iterates of each iteration sequence (over the Parikh and language semirings). We use Newton's iteration instead of the usual Kleene's iteration sequence because Newton's iteration is guaranteed to converge on the Parikh semiring in a finite number of steps, a property that we shall exploit. Kleene's iteration sequence, on the other hand, may not converge. Due to lack of space, we refer the reader to [4,5] for the definition of Newton's iteration scheme.

We first extend the definition of the Parikh image to a valuation $\boldsymbol{v} \in \mathcal{L}^{\mathcal{X}}$ as the valuation of $\mathcal{P}^{\mathcal{X}}$ defined for each variable X by: $\Pi(\boldsymbol{v})(X) = \Pi(\boldsymbol{v}(X))$. Then, given $\boldsymbol{F}_{\mathcal{L}} \colon \mathcal{L}^{\mathcal{X}} \to \mathcal{L}^{\mathcal{X}}$, a polynomial transformation, we define a polynomial transformation $\boldsymbol{F}_{\mathcal{P}} \colon \mathcal{P}^{\mathcal{X}} \to \mathcal{P}^{\mathcal{X}}$ such that: for every $X \in \mathcal{X}$ we have $\boldsymbol{F}_{\mathcal{P}X} = \Pi \circ \boldsymbol{F}_{\mathcal{L}X} \circ \Pi^{-1}$. Lemma. 1 relates the iterates for $\mu\boldsymbol{F}_{\mathcal{L}}$ and $\mu\boldsymbol{F}_{\mathcal{P}}$ using the Parikh image mapping.

Lemma 1. *Let $(\boldsymbol{\nu}_i)_{i\in\mathbb{N}}$ and $(\boldsymbol{\kappa}_i)_{i\in\mathbb{N}}$ be Newton's iteration sequences associated with $\boldsymbol{F}_{\mathcal{L}}$ and $\boldsymbol{F}_{\mathcal{P}}$, respectively. For every $i \in \mathbb{N}$, we have $\Pi(\boldsymbol{\nu}_i) = \boldsymbol{\kappa}_i$.*

In [5], the authors show that Newton's iterates converges after a finite number of steps when defined over a commutative ω-continuous semiring. This shows, in our setting, that $(\boldsymbol{\kappa}_i)_{i\in\mathbb{N}}$ stabilizes after a finite number of steps.

Lemma 2. *Let $(\boldsymbol{\kappa}_i)_{i\in\mathbb{N}}$ be Newton's iteration sequence associated to $\boldsymbol{F}_{\mathcal{P}}$ and let n be the number of variables in $\mathcal{X}$. For every $k \geq n$, we have $\boldsymbol{\kappa}_k = \Pi(\mu\boldsymbol{F}_{\mathcal{L}})$. Hence, for every $k \geq n$, $\Pi(\boldsymbol{\nu}_k) = \Pi(\mu\boldsymbol{F}_{\mathcal{L}})$.*

We know Newton's iteration sequence $(\boldsymbol{\nu}_i)_{i\in\mathbb{N}}$, whose limit is $\mu\boldsymbol{F}_{\mathcal{L}}$, may not converge after a finite number of iterations. However, using Lem. 2, we know that the Parikh image of the iterates stabilizes after a finite number of steps. Precisely, if n is the number of variables in $\mathcal{X}$, then the language given by $\boldsymbol{\nu}_n$ is

such that $\Pi(\boldsymbol{\nu}_n) = \Pi(L(G))$. Moreover because $(\boldsymbol{\nu}_i)_{i\in\mathbb{N}}$ is an ascending chain, for each variable $X \in \mathcal{X}$, we have that $\boldsymbol{\nu}_n(X)$ is a sublanguage of $L_X(G)$ such that $\Pi(\boldsymbol{\nu}_n(X)) = \Pi(L_X(G))$.

3.2 Representation of Iterates

We now show that Newton's iterates can be effectively represented as a combination of linear grammars and homomorphisms.

A *substitution* σ from alphabet Σ_1 to alphabet Σ_2 is a function which maps every word over Σ_1 to a set of words of Σ_2^* such that $\sigma(\varepsilon) = \{\varepsilon\}$ and $\sigma(u \cdot v) = \sigma(u) \cdot \sigma(v)$. A *homomorphism* h is a substitution such that for each word u, $h(u)$ is a singleton. We define the substitution $\sigma_{[a/b]} \colon \Sigma_1 \cup \{a\} \to \Sigma_1 \cup \{b\}$ which maps a to b and leaves all other symbols unchanged.

We show below that the iterates $(\boldsymbol{\nu}_k)_{k\leq n}$ have a "nice" representation.

k-fold composition. We effectively compute and represent each iterate as the valuation which maps each variable X to the language generated by a k-fold composition of a substitution. Since the substitution maps each symbol onto a language which is linear, it is effectively represented and manipulated as a linear grammar. To formally define the representation we need to introduce the following definitions.

Let $\tilde{G} = (\mathcal{X}, \Sigma \cup \{v_X \mid X \in \mathcal{X}\}, \tilde{\delta})$ be a linear grammar and let $k \in \mathbb{N}$, define $v_{\mathcal{X}}^k$ to be the set of symbols $\{v_X^k \mid X \in \mathcal{X}\}$. Given a language L on alphabet $\Sigma \cup \{v_X \mid X \in \mathcal{X}\}$, we define $L[v_{\mathcal{X}}^k]$ to be $\sigma_{[v_X/v_X^k]_{X\in\mathcal{X}}}(L)$ that is the language where each occurence of v_X is replaced by v_X^k.

For $k \in \{1, \dots, n\}$, we define $\sigma_k \colon \Sigma \cup v_{\mathcal{X}}^k \to \Sigma \cup v_{\mathcal{X}}^{k-1}$ as the substitution which maps each v_X^k onto $L_X(\tilde{G})[v_{\mathcal{X}}^{k-1}]$ and leaves Σ unchanged. For $k = 0$ the substitution σ_0 maps each v_X^0 on $\boldsymbol{F}(\ddot{0})(X)$ and leaves Σ unchanged. σ_0 basically applies the terminal rules of the grammar. Let k, ℓ be such that $0 \leq k \leq \ell \leq n$ we define σ_k^ℓ to be $\sigma_k \circ \cdots \circ \sigma_\ell$. Hence, σ_0^k is such that: $(\Sigma \cup v_{\mathcal{X}}^k)^* \xrightarrow{\sigma_k} (\Sigma \cup v_{\mathcal{X}}^{k-1})^* \cdots (\Sigma \cup v_{\mathcal{X}}^1)^* \xrightarrow{\sigma_1} (\Sigma \cup v_{\mathcal{X}}^0)^* \xrightarrow{\sigma_0} \Sigma^*$.

Finally, the *k-fold composition* of a linear grammar $\tilde{G}$ and initial variable X is given by $\sigma_0^k(v_X^k)$. Lemma 3 relates k-fold compositions with $(\boldsymbol{\nu}_k)_{k\in\mathbb{N}}$. Moreover we characterize the complexity of computing $\tilde{G}$ given a polynomial transformation $\boldsymbol{F}$ the size of which is defined to be the number of bits needed to write the set $\{\boldsymbol{F}_X\}_{X\in\mathcal{X}}$ where each $\boldsymbol{F}_X$ is a string of symbols.

Lemma 3. *Given a polynomial transformation $\boldsymbol{F}$, there is a polynomial time algorithm to compute a linear grammar $\tilde{G}$ such that for every $k \geq 0$, every $X \in \mathcal{X}$ we have $\boldsymbol{\nu}_k(X) = \sigma_0^k(v_X^k)$.*

We refer the reader to our technical report [7] for the polynomial time construction of $\tilde{G}$ given $\boldsymbol{F}$. However, let us give a sample output of the construction.

Example 2. Let $\boldsymbol{F}$ be a polynomial transformation on $\mathcal{L}^{\mathcal{X}}$ where $\boldsymbol{F}_{X_0} = aX_1 \cup a$ and $\boldsymbol{F}_{X_1} = X_0 b \cup aX_1 bX_0$. The construction outputs $\tilde{G} = (\{X_0, X_1\}, \{a, b, v_{X_0}, v_{X_1}\}, \tilde{\delta})$ where $\tilde{\delta}$ is given by:

$$X_0 \to aX_1 \mid av_{X_1} \mid a$$
$$X_1 \to X_0 b \mid aX_1 bv_{X_0} \mid av_{X_1} bX_0 \mid v_{X_0} b \mid av_{X_1} bv_{X_0} \ .$$

We have that $\boldsymbol{\nu}_1(X_0) = \sigma_0 \circ \sigma_1(v^1_{X_0})$ and $\boldsymbol{\nu}_1(X_1) = \sigma_0 \circ \sigma_1(v^1_{X_1})$.

Lem. 3 completes our goal to define a procedure to effectively compute and represent the iterates $(\boldsymbol{\nu}_k)_{k\in\mathbb{N}}$. This sequence is of interest since, given a CFL L and $\boldsymbol{\nu}_n$ the n-th iterate (where n equals the number of variables in the grammar of L so that $\Pi(\boldsymbol{\nu}_n) = \Pi(L)$), if B is a solution to Pb. 1 for the instance $\boldsymbol{\nu}_n$, B is also a solution to Pb. 1 for L. Notice that k-fold compositions relate to indexed grammars used to represent Newton iterates in [6].

4 Constructing a Parikh Equivalent Bounded Subset

We now show how, given a k-fold composition L', to compute an elementary bounded language B such that $\Pi(L' \cap B) = \Pi(B)$, that is we give an effective procedure to solve Pb. 1 for the instance L'. This will complete the solution to Pb. 1, hence the proof of Th. 1. In this section, we give an effective construction of elementary bounded languages that solve Pb. 1 first for regular languages, then for linear languages, and finally for a linear substitution.

First we need to introduce the notion of semilinear sets. A set $A \subseteq \mathbb{N}^n$ is a *linear set* if there exist $c \in \mathbb{N}^n$ and $p_1, \ldots, p_k \in \mathbb{N}^n$ such that $A = \left\{ c + \sum_{i=1}^{k} \lambda_j p_j \mid \lambda_j \in \mathbb{N} \right\}$: c is called the constant of A and $p_1, \ldots, p_k$ the periods of A. A *semilinear set* S is a finite union of linear sets: $S = \bigcup_{i=1}^{\ell} A_i$ where each A_i is a linear set. Parikh's theorem (cf. [8]) shows that the Parikh image of every CFL is a semilinear set that is effectively computable.

Lemma 4. *Let L and B be respectively a CFL and an elementary bounded language over Σ such that $\Pi(L \cap B) = \Pi(L)$. There is an effectively computable elementary bounded language B' such that $\Pi(L^t \cap B') = \Pi(L^t)$ for all $t \in \mathbb{N}$.*

Proof. By Parikh's theorem, we know that $\Pi_\Sigma(L)$ is a computable semilinear set. Let us consider $u_1, \ldots, u_\ell \in L$ such that $\Pi_\Sigma(u_i) = c_i$ for $i \in \{1, \ldots, \ell\}$.

Let $B' = u_1^* \cdots u_\ell^* B^\ell$, we see that B' is an elementary bounded language. Let $t > 0$ be a natural integer. We have to prove that $\Pi(L^t) \subseteq \Pi(L^t \cap B')$.

case $t \leq \ell$. By property of Π and $\Pi(L) = \Pi(L \cap B)$ we find that:

$$\begin{aligned}
\Pi(L^t) &= \Pi((L \cap B)^t) \\
&\subseteq \Pi(L^t \cap B^t) && \text{monotonicity of } \Pi \\
&\subseteq \Pi(L^t \cap B^\ell) && B^t \subseteq B^\ell \text{ since } \varepsilon \in B \\
&\subseteq \Pi(L^t \cap B') && \text{def. of } B'
\end{aligned}$$

case $t > \ell$. Let us consider $w \in L^t$. For every $i \in \{1, \ldots, \ell\}$ and $j \in \{1, \ldots, k_i\}$, there exist some positive integers λ_{ij} and μ_i, with $\sum_{i=1}^{\ell} \mu_i = t$ such that

$$\Pi(w) = \sum_{i=1}^{\ell} \mu_i c_i + \sum_{i=1}^{\ell} \sum_{j=1}^{k_i} \lambda_{ij} p_{ij} \ .$$

We define a new variable for each $i \in \{1, \dots, \ell\}$: $\alpha_i = \begin{cases} \mu_i - 1 & \text{if } \mu_i > 0 \\ 0 & \text{otherwise.} \end{cases}$.
For each $i \in \{1, \dots, \ell\}$, we also consider z_i a word of $L \cup \{\varepsilon\}$ such that $z_i = \varepsilon$ if $\mu_i = 0$ and $\Pi(z_i) = c_i + \sum_{j=1}^{k_i} \lambda_{ij} p_{ij}$ else.

Let $w' = u_1^{\alpha_1} \dots u_\ell^{\alpha_\ell} z_1 \dots z_\ell$. Clearly, $\Pi(w') = \Pi(w)$ and $w' \in u_1^* \cdots u_\ell^* (L \cup \{\varepsilon\})^\ell$. For each $i \in \{1, \dots, \ell\}$, $\Pi(L \cap B) = \Pi(L)$ shows that there is $z_i' \in (L \cap B) \cup \{\varepsilon\}$ such that $\Pi(z_i') = \Pi(z_i)$. Let $w'' = u_1^{\alpha_1} \dots u_\ell^{\alpha_\ell} z_1' \dots z_\ell'$. We find that $\Pi(w'') = \Pi(w)$, $w'' \in B'$ and we can easily verify that $w'' \in L^t$. □

Regular Languages. The construction of an elementary bounded language that solves Pb. 1 for a regular language L is known from [13] (see also [14], Lem. 4.1). The construction is carried out by induction on the structure of a regular expression for L. Assuming $L \neq \emptyset$, the base case (*i.e.* a symbol or ε) is trivially solved. Note that if $L = \emptyset$ then every elementary bounded language B is such that $\Pi(L \cap B) = \Pi(L) = \emptyset$.

The inductive case decomposes into three constructs. Let R_1 and R_2 be regular languages, and B_1 and B_2 the inductively constructed elementary bounded languages such that $\Pi(R_1 \cap B_1) = \Pi(R_1)$ and $\Pi(R_2 \cap B_2) = \Pi(R_2)$.

concatenation. For the instance $R_1 \cdot R_2$, the elementary bounded language $B_1 \cdot B_2$ is such that $\Pi((R_1 \cdot R_2) \cap (B_1 \cdot B_2)) = \Pi(R_1 \cdot R_2)$;

union. For $R_1 \cup R_2$, the elementary bounded language $B_1 \cdot B_2$ suffices;

Kleene star. Let us consider R_1 and B_1, Lem. 4 shows how to effectively compute an elementary bounded language B' such that for every $t \in \mathbb{N}$, $\Pi(R_1^t \cap B') = \Pi(R_1^t)$. Let us prove that B' solves Pb. 1 for the instance R_1^*. In fact, if w is a word of R_1^*, there exists a $t \in \mathbb{N}$ such that $w \in R_1^t$. Then, we can find a word w' in $R_1^t \cap B'$ with the same Parikh image as w. This proves that $\Pi(R_1^*) \subseteq \Pi(R_1^* \cap B')$. The other inclusion holds trivially.

Proposition 2. *For every regular language R, there is an effective procedure to compute an elementary bounded language B such that $\Pi(R \cap B) = \Pi(R)$.*

Linear Languages. We now extend the previous construction to the case of linear languages. Recall that linear languages are used to represent the iterates $(\boldsymbol{\nu}_k)_{k \in \mathbb{N}}$. Lemma 5 gives a characterization of linear languages based on regular languages, homomorphism, and some additional structures.

Lemma 5. *(from [10]) For every linear language L over Σ, there exist an alphabet A and its distinct copy $\widetilde{A}$, an homomorphism $h : (A \cup \widetilde{A})^* \to \Sigma^*$ and a regular language R over A such that $L = h(R\widetilde{A}^* \cap S)$ where $S = \{w\widetilde{w}^r \mid w \in A^*\}$ and w^r denotes the reverse image of the word w. Moreover there is an effective procedure to construct h, A, and R.*

The next result shows that an elementary bounded language that solves Pb. 1 can be effectively constructed for every linear language L that is given by h and R such that $L = h(R\widetilde{A}^* \cap S)$.

Proposition 3. *For every linear language $L = h(R\widetilde{A}^* \cap S)$ where h and R are given, there is an effective procedure which solves Pb. 1 for the instance L, that is a procedure returning an elementary bounded B such that $\Pi(L \cap B) = \Pi(L)$.*

Linear languages with Substitutions. Our goal is to solve Pb. 1 for k-fold compositions, *i.e.* for languages of the form $\sigma_j^k(v_X^k)$. Prop. 3 gives an effective procedure for the case $j = k$ since $\sigma_k^k(v_X^k)$ is a linear language. Prop. 4 generalizes to the case $j < k$: given a solution to Pb. 1 for the instance $\sigma_{j+1}^k(v_X^k)$, there is an effective procedure for Pb. 1 for the instance $\sigma_j \circ \sigma_{j+1}^k(v_X^k) = \sigma_j^k(v_X^k)$.

Proposition 4. *Let*

1. *L be a CFL over Σ;*
2. *B an elementary bounded language such that $\Pi(L \cap B) = \Pi(L)$;*
3. *σ and τ be two substitutions over Σ such that for each $a \in \Sigma$, (i) $\sigma(a)$ and $\tau(a)$ are respectively a CFL and an elementary bounded and (ii) $\Pi(\sigma(a) \cap \tau(a)) = \Pi(\sigma(a))$.*

Then, there is an effective procedure that solves Pb. 7 for the instance $\sigma(L)$, by returning an elementary bounded language B' such that $\Pi(\sigma(L) \cap B') = \Pi(\sigma(L))$.

We use the above result inductively to solve Pb. 1 for k-fold composition as follows: fix L to be $\sigma_{j+1}^k(v_X^k)$, B to be the solution of Pb. 1 for the instance L, σ to be σ_j and τ a substitution which maps every v_X^j to the solution of Pb. 1 for the instance $\sigma_j(v_X^j)$. Then B' is the solution of Pb. 1 for the instance $\sigma_j^k(v_X^k)$.

Due to lack of space we refer to reader to [7] for details.

We thus have an effective construction of an elementary bounded language that solves Pb. 1 for k-fold composition, hence a constructive proof for Th. 1.

Iterative Algorithm. We conclude this section by showing a result related to the notion of progress if the result of Th. 1 is applied repeatedly.

Lemma 6. *Given a CFL L, define two sequences $(L_i)_{i\in\mathbb{N}}$, $(B_i)_{i\in\mathbb{N}}$ such that (1) $L_0 = L$, (2) B_i is elementary bounded and $\Pi(L_i \cap B_i) = \Pi(L_i)$, (3) $L_{i+1} = L_i \cap \overline{B_i}$. For every $w \in L$, there exists $i \in \mathbb{N}$ such that $w \notin L_i$. Moreover, given L_0, there is an effective procedure to compute L_i for every $i > 0$.*

Proof. Let $w \in L$ and let $v = \Pi(w)$ be its Parikh image. We conclude from $\Pi(L_0 \cap B_0) = \Pi(L_0)$ that there exists a word $w' \in B_0$ such that $\Pi(w') = v$. Two cases arise: either $w' = w$ and we are done; or $w' \neq w$. In that case $L_1 = L_0 \cap \overline{B_0}$ shows that $w' \notin L_1$. Intuitively, at least one word with the same Parikh image as w has been selected by B_0 and then removed from L_0 by definition of L_1. Repeatedly applying the above reasoning shows that at each iteration there exists a word w'' such that $\Pi(w'') = v$, $w'' \in B_i$ and $w'' \notin L_{i+1}$ since $L_{i+1} = L_i \cap \overline{B_i}$. Because there are only finitely many words with Parikh image v we conclude that there exists $j \in \mathbb{N}$, such that $w \notin L_j$. The effectiveness result follows from the following arguments: (1) as we have shown above (our solution to Pb. 1), given

a CFL L there is an effective procedure that computes an elementary bounded language B such that $\Pi(L \cap B) = \Pi(L)$; (2) the complement of B is a regular language effectively computable; and (3) the intersection of a CFL with a regular language is again a CFL that can be effectively constructed (see [10]). □

Intuitively this result shows that given a context-free language L, if we repeatedly compute and remove a Parikh-equivalent bounded subset of L ($L \cap \overline{B}$ is effectively computable since B is a regular language), then each word w of L is eventually removed from it.

5 Application to Multithreaded Procedural Programs

We now give an application of our construction that gives a semi-algorithm for checking reachability of multithreaded procedural programs [16,11,2]. A common programming model consists of multiple recursive threads communicating via shared memory. Formally, we model such systems as pushdown networks [17]. Let k be a positive integer, a *pushdown network* is a triple $\mathcal{N} = (G, \Gamma, (\Delta_i)_{1 \leq i \leq k})$ where G is a finite non-empty set of *globals*, Γ is the *stack alphabet*, and for each $1 \leq i \leq k$, Δ_i is a finite set of *transition rules* of the form $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$ for $g, g' \in G$, $\gamma \in \Gamma$, $\alpha \in \Gamma^*$.

A *local configuration* of $\mathcal{N}$ is a pair $(g, \alpha) \in G \times \Gamma^*$ and a *global configuration* of $\mathcal{N}$ is a tuple $(g, \alpha_1, \ldots, \alpha_k)$, where $g \in G$ and $\alpha_1, \ldots, \alpha_k \in \Gamma^*$ are individual stack content for each thread. Intuitively, the system consists of k threads, each of which with its own stack, and the threads can communicate by reading and manipulating the global storage represented by g.

We define the local transition relation of the i-th thread, written $\rightarrow_i$, as follows: $(g, \gamma\beta) \rightarrow_i (g', \alpha\beta)$ iff $\langle g, \gamma \rangle \hookrightarrow \langle g', \alpha \rangle$ in Δ_i and $\beta \in \Gamma^*$. The transition relation of $\mathcal{N}$, denoted $\rightarrow$, is defined as follows: $(g, \alpha_1, \ldots, \alpha_i, \ldots, \alpha_k) \rightarrow (g', \alpha_1, \ldots, \alpha'_i, \ldots, \alpha_k)$ iff $(g, \alpha_i) \rightarrow_i (g', \alpha'_i)$ for some $i \in \{1, \ldots, k\}$. By $\rightarrow_i^*$, $\rightarrow^*$, we denote the reflexive and transitive closure of these relations. Let C_0 and C be two global configurations, the *reachability problem* asks whether $C_0 \rightarrow^* C$ holds. An instance of the reachability problem is denoted by a triple $(\mathcal{N}, C_0, C)$.

A *pushdown system* is a pushdown network where $k = 1$, namely (G, Γ, Δ). A *pushdown acceptor* is a pushdown system extended with an initial configuration $c_0 \in G \times \Gamma^*$, labeled transition rules of the form $\langle g, \gamma \rangle \overset{\lambda}{\hookrightarrow} \langle g'\alpha \rangle$ for g, g', γ, α defined as above and $\lambda \in \Sigma \cup \{\varepsilon\}$. A pushdown acceptor is given by a tuple $(G, \Gamma, \Sigma, \Delta, c_0)$. The language of a pushdown acceptor is defined as expected where the acceptance condition is given by the empty stack.

In what follows, we reduce the reachability problem for a pushdown network of k threads to a language problem for k pushdown acceptors. The pushdown acceptors obtained by reduction from the pushdown network settings have a special global $\perp$ that intuitively models an inactive state. The reduction also turns the globals into input symbols which label transitions. The firing of a transition labeled with a global models a context switch. When such transition fires, every pushdown acceptor synchronizes on the label. The effect of such a

synchronization is that exactly one acceptor will change its state from inactive to active by updating the value of its global (i.e. from $\bot$ to some $g \in G$) and exactly one acceptor will change from active to inactive by updating its global from some g to $\bot$. All the others acceptors will synchronize and stay inactive.

Given an instance of the reachability problem, that is a pushdown network $(G, \Gamma, (\Delta_i)_{1\leq i\leq k})$ with k threads, two global configurations C_0 and C (assume wlog that C is of the form $(g, \varepsilon, \ldots, \varepsilon)$), we define a family of pushdown acceptors $\{(G', \Gamma, \Sigma, \Delta'_i, c_0^i)\}_{1\leq i\leq k}$, where:

- $G' = G \cup \{\bot\}$, Γ is given as above, and $\Sigma = G \times \{1, \ldots, k\}$,
- Δ'_i is the smallest set such that:
  - $\langle g, \gamma\rangle \overset{\varepsilon}{\hookrightarrow} \langle g', \alpha\rangle$ in Δ'_i if $\langle g, \gamma\rangle \hookrightarrow \langle g', \alpha\rangle$ in Δ_i;
  - $\langle g, \gamma\rangle \overset{(g,j)}{\hookrightarrow} \langle \bot, \gamma\rangle$ for $j \in \{1, \ldots, k\} \setminus \{i\}$, $g \in G$, $\gamma \in \Gamma$;
  - $\langle \bot, \gamma\rangle \overset{(g,j)}{\hookrightarrow} \langle \bot, \gamma\rangle$ for $j \in \{1, \ldots, k\} \setminus \{i\}$, $g \in G$, $\gamma \in \Gamma$;
  - $\langle \bot, \gamma\rangle \overset{(g,i)}{\hookrightarrow} \langle g, \gamma\rangle$ for $g \in G$, $\gamma \in \Gamma$.
- let $C_0 = (g, \alpha_1, \ldots, \alpha_i, \ldots, \alpha_k)$, c_0^i is given by $(\bot, \alpha_i)$ if $i > 1$; (g, α_1) else.

Proposition 5. *Let k be a positive integer, and $(\mathcal{N}, C_0, C)$ be an instance of the reachability problem with k threads, one can effectively construct CFLs $(L_1, \ldots, L_k)$ (as pushdown acceptors) such that $C_0 \to^* C$ iff $L_1 \cap \cdots \cap L_k \neq \emptyset$.*

The converse of the proposition is also true, and since the emptiness problem for intersection of CFLs is undecidable [10], so is the reachability problem. We will now compare two underapproximation techniques for the reachability problem: context-bounded switches [15] and bounded languages, which we first detail below.

Let $L_1, \ldots, L_k$ be context-free languages, and consider the problem to decide if $\bigcap_{1\leq i\leq k} L_i \neq \emptyset$. We give a decidable sufficient condition: given an elementary bounded language B, we define the *intersection modulo B* of the languages $\{L_i\}_i$ as $\bigcap_i^{(B)} L_i = \big(\bigcap_i L_i\big) \cap B$. Clearly, $\bigcap_i^{(B)} L_i \neq \emptyset$ implies $\bigcap_i L_i \neq \emptyset$. Below we show that the problem $\bigcap_i^{(B)} L_i \neq \emptyset$ is decidable .

Lemma 7. *Given an elementary bounded language $B = w_1^* \cdots w_n^*$ and CFLs $L_1, \ldots, L_k$, it is decidable to check if $\bigcap_{1\leq i\leq k}^{(B)} L_i \neq \emptyset$.*

Proof. Define the alphabet $A = \{a_1, \ldots, a_n\}$ disjoint from Σ. Let h be the homomorphism that maps the symbols $a_1, \ldots, a_n$ to the words $w_1, \ldots, w_n$, respectively. We show that $\bigcap_{1\leq i\leq k} \Pi_A\big(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*\big) \neq \emptyset$ iff $\bigcap_{1\leq i\leq k}^{(B)} L_i \neq \emptyset$.

We conclude from $w \in \bigcap_{1\leq i\leq k}^{(B)} L_i$ that $w \in B$ and $w \in L_i$ for every $1 \leq i \leq k$, hence there exist $t_1, \ldots, t_n \in \mathbb{N}$ such that $w = w_1^{t_1} \ldots w_n^{t_n}$ by definition of B. Then, we find that $(t_1, \ldots, t_n) \in \Pi_A(h^{-1}(w) \cap a_1^* \cdots a_n^*)$, hence that $(t_1, \ldots, t_n) \in \Pi_A(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*)$ for every $1 \leq i \leq k$ by above and finally that $(t_1, \ldots, t_n) \in \bigcap_{1\leq i\leq k} \Pi_A\big(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*\big)$.

For the other implication, consider $(t_1, \ldots, t_n)$ a vector of $\bigcap_{1\leq i\leq k} \Pi_A\big(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*\big)$ and let $w = w_1^{t_1} \ldots w_n^{t_n}$. For every $1 \leq i \leq k$,

we will show that $w \in L_i \cap B$. As $(t_1, \ldots, t_n) \in \Pi_A\big(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*\big)$, there exists a word $w' \in a_1^* \cdots a_n^*$ such that $\Pi_A(w') = (t_1, \ldots, t_n)$ and $h(w') \in L_i \cap B$. We conclude from $\Pi_A(w') = (t_1, \ldots, t_n)$, that $w' = a_1^{t_1} \ldots a_n^{t_n}$ and finally that, $h(w') = w$ belongs to $L_i \cap B$.

The class of CFLs is effectively closed under inverse homomorphism and intersection with a regular language [10]. Moreover, given a CFL, we can compute its Parikh image which is a semilinear set. Finally, we can compute the semilinear sets $\Pi_A\big(h^{-1}(L_i \cap B) \cap a_1^* \cdots a_n^*\big)$ and the emptiness of the intersection of semilinear sets is decidable [8]. □

While Lem. 7 shows decidability for every elementary bounded language, in practice, we want to select B "as large as possible". We select B using Th. 1. We first compute for each language L_i the elementary bounded language $B_i = w_1^{(i)^*} \cdots w_{n_i}^{(i)^*}$ such that $\Pi(L_i \cap B_i) = \Pi(L_i)$. Finally, we choose $B = B_1 \cdots B_k$.

By repeatedly selecting and removing a bounded language B from each L_i where $1 \leq i \leq k$ we obtain a sequence $\{L_i^j\}_{j \geq 0}$ of languages such that $L_i = L_i^0 \supseteq L_i^1 \supseteq \ldots$ The result of Lem. 6 shows that for each word $w \in L_i$, there is some j such that $w \notin L_i^j$, hence that the above sequence is strictly decreasing, that is $L_i = L_i^0 \supsetneq L_i^1 \supsetneq \ldots$, and finally that if $\bigcap_{1 \leq i \leq k} L_i \neq \emptyset$ then the iteration is guaranteed to terminate.

Comparison with Context-Bounded Reachability. A well-studied under-approximation for multithreaded reachability is given by context-bounded reachability [15]. We need a few preliminary definitions. We define the global reachability relation $\rightsquigarrow$ as a reachability relation where all the moves are made by a single thread: $(g, \alpha_1, \ldots, \alpha_i, \ldots, \alpha_n) \rightsquigarrow (g', \alpha_1, \ldots, \alpha_i', \ldots, \alpha_n)$ iff $(g, \alpha_i) \rightarrow_i^* (g', \alpha_i')$ for some $1 \leq i \leq n$. The relation $\rightsquigarrow$ holds between global configurations reachable from each other in a single *context*. Furthermore we denote by $\rightsquigarrow_j$, where $j \geq 0$, the reachability relation within j contexts: $\rightsquigarrow_0$ is the identity relation on global configurations, and $\rightsquigarrow_{i+1} = \rightsquigarrow_i \circ \rightsquigarrow$.

Given a pushdown network, global configurations C_0 and C, and a number $k \geq 1$, the *context-bounded reachability problem* asks whether $C_0 \rightsquigarrow_k C$ holds, i.e. if C can be reached from C_0 in k context switches. This problem is decidable [15]. Context-bounded reachability has been successfully used in practice for bug finding. We show that underapproximations using bounded languages (Lem. 7) subsumes the technique of context-bounded reachability in the following sense.

Proposition 6. *Let $\mathcal{N}$ be a pushdown network, C_0, C global configurations of $\mathcal{N}$, and $(L_1, \ldots, L_n)$ CFLs over alphabet Σ such that $C_0 \rightarrow^* C$ iff $\cap_i L_i \neq \emptyset$. For each $k \geq 1$, there is an elementary bounded language B_k such that $C_0 \rightsquigarrow_k C$ only if $\bigcap_i^{(B_k)} L_i \neq \emptyset$. Also, $\bigcap_i^{(B_k)} L_i \neq \emptyset$ only if $C_0 \rightarrow^* C$.*

Proof. Consider all sequences $C_0 \rightsquigarrow C_1 \cdots C_{k-1} \rightsquigarrow C_k$ of k or fewer switches. By the CFL encoding (Prop. 5) each of these sequences corresponds to a word in Σ^k. If $C_0 \rightsquigarrow_k C$, then there is a word $w \in \bigcap_i L_i$ and $w \in \Sigma^k$. Define B_k to be $w_1^* \cdots w_m^*$ where $w_1, \ldots, w_m$ is an enumeration of all strings in Σ^k. We conclude from $w \in \Sigma^k$ and the definition of B_k that $w \in B_k$, hence that $\bigcap_i^{(B_k)} L_i \neq \emptyset$

```
thread p1() {
  int c=0;
L:bit=true;
  if bit == false { ++c; }
  if c<k { goto L; }
}
```

```
thread p2() {
L1:bit = false;
   goto L1;
}
```

Fig. 1. The family of pushdown network with global `bit`

since $w \in \bigcap_i L_i$. For the other direction we conclude from $\bigcap_i^{(B_k)} L_i \neq \emptyset$ that $\bigcap_i L_i \neq \emptyset$, hence that $C_0 \rightarrow^* C$. □

However, underapproximation using bounded languages can be more powerful than context-bounded reachability in the following sense. There is a family $\{(\mathcal{N}_k, C_{0k}, C_k)\}_{k\in\mathbb{N}}$ of pushdown network reachability problems such that $C_{0k} \rightsquigarrow_k C_k$ but $C_{0k} \not\rightsquigarrow_{k-1} C_k$ for each k, but there is a single elementary bounded B such that $\bigcap_i^{(B)} L_{ik} \neq \emptyset$ for each k, where again $(L_{1k}, \ldots, L_{nk})$ are CFLs such that $C_{0k} \rightsquigarrow C_k$ iff $\cap_i L_{ik} \neq \emptyset$ (as in Prop. 5).

For clarity, we describe the family of pushdown networks as a family of two-threaded programs whose code is shown in Fig. 1. The programs in the family differs from each other by the value to which `k` is instantiated: `k` $= 0, 1, \ldots$ Each program has two threads. Thread one maintains a local counter `c` starting at `0`. Before each increment to `c`, thread one sets a global `bit`. Thread two resets `bit`. The target configuration C_k is given by the exit point of `p1`. We conclude from the program code that hitting the exit point of `p1` requires `c` $\geq$ `k` to hold. For every instance, C_k is reachable, but it requires at least k context switches. Thus, there is no fixed context bound that is sufficient to check reachability for every instance in the family. In contrast, the elementary bounded language given by $\big((\texttt{bit} == \texttt{true}, 2) \cdot (\texttt{bit} == \texttt{false}, 1)\big)^*$ is sufficient to show reachability of the target for **every** instance in the family.

Acknowledgment. We thank Ahmed Bouajjani for pointing that the bounded languages approach subsumes the context-bounded switches one.

References

1. Blattner, M., Latteux, M.: Parikh-bounded languages. In: Even, S., Kariv, O. (eds.) ICALP 1981. LNCS, vol. 115, pp. 316–323. Springer, Heidelberg (1981)
2. Bouajjani, A., Esparza, J., Touili, T.: A generic approach to the static analysis of concurrent programs with procedures. In: POPL'03, pp. 62–73. ACM Press, New York (2003)
3. Chomsky, N., Schützenberger, M.P.: The algebraic theory of context-free languages. In: Comp. Programming and Formal Systems, pp. 118–161. North-Holland, Amsterdam (1963)
4. Esparza, J., Kiefer, S., Luttenberger, M.: An extension of Newton's method to ω-continuous semirings. In: Harju, T., Karhumäki, J., Lepistö, A. (eds.) DLT 2007. LNCS, vol. 4588, pp. 157–168. Springer, Heidelberg (2007)

5. Esparza, J., Kiefer, S., Luttenberger, M.: On fixed point equations over commutative semirings. In: Thomas, W., Weil, P. (eds.) STACS 2007. LNCS, vol. 4393, pp. 296–307. Springer, Heidelberg (2007)
6. Esparza, J., Kiefer, S., Luttenberger, M.: Newton's Method for ω-Continuous Semirings. In: Aceto, L., Damgård, I., Goldberg, L.A., Halldórsson, M.M., Ingólfsdóttir, A., Walukiewicz, I. (eds.) ICALP 2008, Part II. LNCS, vol. 5126, pp. 14–26. Springer, Heidelberg (2008)
7. Ganty, P., Majumdar, R., Monmege, B.: Bounded Underapproximations. In: CoRR (2009) abs/0809.1236
8. Ginsburg, S.: The Mathematical Theory of Context-Free Languages. McGraw-Hill, NY (1966)
9. Ginsburg, S., Spanier, E.: Bounded ALGOL-like languages. Trans. Amer. Math. Soc. 113, 333–368 (1964)
10. Hopcroft, J.E., Ullman, J.D.: Introduction to Automata Theory, Languages, and Computation, 1st edn. Addison Wesley, Reading (1979)
11. Kahlon, V.: Tractable analysis for concurrent programs via bounded languages (unpublished)
12. Kuich, W.: Semirings and formal power series: their relevance to formal languages and automata. In: Handbook of formal languages, vol. 1, pp. 609–677. Springer, Heidelberg (1997)
13. Latteux, M., Leguy, J.: Une propriété de la famille GRE. In: Fundamentals of Computation Theory, pp. 255–261. Akademie-Verlag, Berlin (1979)
14. Leroux, J., Sutre, G.: On flatness for 2-dimensional vector addition systems with states. In: Gardner, P., Yoshida, N. (eds.) CONCUR 2004. LNCS, vol. 3170, pp. 402–416. Springer, Heidelberg (2004)
15. 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)
16. Ramalingam, G.: Context-sensitive synchronization-sensitive analysis is undecidable. ACM TOPLAS 22(2), 416–430 (2000)
17. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic Context-Bounded Analysis of Multithreaded Java Programs. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)

Global Reachability in Bounded Phase Multi-stack Pushdown Systems

Anil Seth

Department of Computer Science & Engg.
I.I.T. Kanpur, Kanpur 208016, India
seth@cse.iitk.ac.in

Abstract. Bounded phase multi-stack pushdown systems (*mpds*) have been studied recently. Given a set $\mathcal{C}$ of configurations of a *mpds* $\mathcal{M}$, let $pre^*_{\mathcal{M}}(\mathcal{C}, k)$ be the set of configurations of $\mathcal{M}$ from which $\mathcal{M}$ can reach an element of $\mathcal{C}$ in at most k phases. In this paper, we show that for any *mpds* $\mathcal{M}$, any regular set $\mathcal{C}$ of configurations of $\mathcal{M}$ and any number k, the set $pre^*_{\mathcal{M}}(\mathcal{C}, k)$, is regular. We use saturation like method to construct a non-deterministic finite multi-automaton recognizing $pre^*_{\mathcal{M}}(\mathcal{C}, k)$. Size of the automaton constructed is double exponential in k which is optimal as the worst case complexity measure.

1 Introduction

Model checking of programs with threads is an important problem that has been considered in several recent works, see [11,7,8,13]. Boolean valued programs with recursion and a fixed number of threads can be modeled as a multi-stack pushdown system (*mpds*). A *mpds* has a finite set of control states and a fixed number of stacks. The transition function of a *mpds* may (nondeterministically) do a push or a pop operation on any stack along with a possible change in control state of *mpds*. Each thread has its own stack for its procedures calls and communication among threads is through the common finite states of *mpds*.

It is well known that even a two stack *pds* can simulate universal models of computation. Therefore to get effectively checkable properties of the model, a restriction called bounded context switching was introduced on *mpds* in [11]. In a k context switching *mpds*, we consider only those runs of *mpds* which can be divided into k stages, where each stage is a consecutive sequence of moves from the run in which push and pop operations are performed only in one stack. In [11], reachability between configurations of k context switching *mpds* is shown to be decidable. In fact [11], extends the techniques of [4] to show that bounded context switching *mpds* admit effective global reachability analysis also. More precisely, [11] shows that for any *mpds* M and any regular set $\mathcal{C}$ of configurations of M, $pre^*(\mathcal{C}, k)$, the set of configurations of M from which a configuration in $\mathcal{C}$ can be reached by M in at most k context switches, is regular. Similarly $post^*(\mathcal{C}, k)$, the set of configurations to which M can reach from some configuration in $\mathcal{C}$, in at most k context switches, is also shown to be regular in [11]. Bounded context

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 615–628, 2010.

switching model checking has been fruitful in uncovering some errors in software and has received some attention in theory also, see [3,10].

In [7], a generalization of bounded context switching *mpds*, called bounded phase *mpds* have been considered. In a $k-$phase bounded *mpds* also a run is divided into k stages but now in one stage while *pop* operations are performed only in one stack, push operations can be performed on any stack. A stage is called a phase in this case. Bounded phase *mpds* capture a strictly larger class of systems than bounded context switching *mpds*. In [7] emptiness problem of bounded phase multi-stack automata (*mpda*) is shown to be decidable. Apart from being interesting in their own right, bounded phase *mpds* can simulate some other interesting systems also. For example, in [8] networks of finite state processes, where processes can send messages to each other via FIFO queues, are studied. There network architectures are presented where (unbounded) reachability between configurations can be reduced to bounded phase reachability. As a more theoretical result, bounded phase multi-stack pushdown transducers have been used to give an infinite automaton characterization of the complexity class of problems solvable in double exponential time, 2ETIME [9]. In [1], a little more general automata model than *mpda* has been presented and its emptiness problem is shown to be decidable.

In this paper we are interested in global model checking of reachability over bounded phase *mpds*. Let $\mathcal{C}$ be a regular set of configurations of an *mpds*. As mentioned above $pre^*(\mathcal{C}, k)$, and $post^*(\mathcal{C}, k)$ are regular if k is the number of context switches. However, if k is the number of phases of *mpds* then $post^*(\mathcal{C}, k)$, is *not* always regular even for $k = 1$. We present an example, modified from an example mentioned at the end of [7], showing this in section 2.3. This leaves the question if $pre^*(\mathcal{C}, k)$ is regular for a regular set $\mathcal{C}$. By results of [1], reachability in a k phase *mpds* can be simulated by reachability in an order-$2k$ higher order pushdown system. This suggests, by the results of [5], that $pre^*(\mathcal{C}, k)$ for a k phase *mpds* can be constructed in time $exp_{2k}(|Q|)$, where exp_n denotes tower of exponents of height n and $|Q|$ is the number of states in the *mpds*. This seems to be the only known method for constructing $pre^*(\mathcal{C}, k)$. In contrast in this paper, we show that $pre^*(\mathcal{C}, k)$ for a k phase *mpds* can be constructed in time $|Q|^{(c.l)^k}$, where c is a constant and l is the number of stacks of the *mpds*. This complexity is only double exponential in k. It is also the optimal worst case complexity bound as the emptiness problem of k-phase *mpda*, which has a lower bound of double exponential in k [9], can be easily reduced to the problem of constructing an automaton recognizing pre^* for a k phase bounded system. The basic idea of our proof is to construct an automaton recognizing pre^* for a single phase and then iterate this construction k-times to get an automaton recognizing pre^* for a k phase bounded system. The construction for single phase uses a modification of saturation technique of [4] and some ideas from [12].

Let $\mathcal{A}_{pre,k}$ be the finite automaton constructed to accept $pre^*(\mathcal{C}, k)$ for a given *mpds* M and regular set C. We also give an algorithm which from any accepting run of $\mathcal{A}_{pre,k}$ on configuration d constructs a witnessing execution sequence s of transitions of M starting at d and ending at some $e \in \mathcal{C}$.

2 Preliminaries

2.1 Multi-stack Pushdown System

Definition 1. *A multi-stack pushdown system (mpds) is given as a tuple* $(Q, \Gamma, l, \delta, q_0)$*, where* Q *is a finite set of states,* l *is the number of stacks,* Γ *is the stack alphabet and* q_0 *is the initial state. The transition function* δ *is given as* $\delta = \delta_{ins} \cup \delta_{rem} \cup \delta_{exch}$*, where*

- $\delta_{exch} \subseteq Q \times \Gamma \times Q \times [1 \dots l] \times \Gamma$,
- $\delta_{ins} \subseteq Q \times \Gamma \times Q \times [1 \dots l] \times \Gamma$,
- $\delta_{rem} \subseteq Q \times \Gamma \times Q \times [1 \dots l]$

($\delta_{exch}, \delta_{ins}, \delta_{rem}$ *represent exchange, push and pop operations respectively). An* mpds *operation depends on its control state and topmost symbol of the stack on which the operation is done.*

Definition 2. *A configuration of multi-stack pushdown system* (Q, Γ, l, δ) *is a tuple* $(q, s_1, \dots, s_l)$*, where* $q \in Q$ *and* $s_i \in \Gamma^*$*, for* $1 \leq i \leq l$*. One step transition* $\xrightarrow{t}$ *on configurations of mpds is defined as below.*

- $(q, s_1, \dots, s_l) \xrightarrow{t} (q', s'_1, \dots, s'_l)$ *if* $t = (q, \gamma, q', i, \gamma') \in \delta_{exch}$, $s_i = \gamma.z_i$, $s'_i = \gamma'.z_i$ *for some* $z_i \in \Gamma^*$ *and* $s'_j = s_j$ *for* $j \neq i$, $1 \leq j \leq l$.
- $(q, s_1, \dots, s_l) \xrightarrow{t} (q', s'_1, \dots, s'_l)$ *if* $t = (q, \gamma, q', i, \gamma') \in \delta_{ins}$, $s_i = \gamma.z_i$, $s'_i = \gamma'.\gamma.z_i$ *for some* $z_i \in \Gamma^*$ *and* $s'_j = s_j$ *for* $j \neq i$, $1 \leq j \leq l$.
- $(q, s_1, \dots, s_l) \xrightarrow{t} (q', s'_1, \dots, s'_l)$ *if* $t = (q, \gamma, q', i) \in \delta_{rem}$, $s_i = \gamma.s'_i$ *and* $s'_j = s_j$ *for* $j \neq i$, $1 \leq j \leq l$.

Our stacks grow from right to left. In the above definition of *mpds,* we do not provide for bottom markers of the stacks explicitly. In fact, we allow a stack to be empty. If stack-i is empty, we represent it by ϵ (the empty string). There is no top of the stack symbol in empty stack-i, we use the convention that $\gamma_i = \epsilon$ in this case. It should be clear from the transition function definition the no *push* or *pop* operation is permitted on an empty stack. However, *push* and *pop* operations on other (non-empty) stacks may still take place.

If a bottom marker is needed, it can be taken as any symbol in Γ with some restriction on transitions involving it.

Definition 3. *A multi-step transition between configurations of mpds, on say sequence* $t_1 t_2 \dots t_n$ *of* mpds *moves,* $c \xrightarrow{t_1 t_2 \dots t_n} d$ *is defined as follows.* $c \xrightarrow{t_1 t_2 \dots t_n} d$ *iff either* $n = 0$ *and* $c = d$ *or there is a* c' *s.t.* $c \xrightarrow{t_1} c'$ *and* $c' \xrightarrow{t_2 \dots t_n} d$*. We write* $c \twoheadrightarrow d$ *for a multi-step transition from* c *to* d *when the sequence of* mpds *moves is not relevant.*

Definition 4. *A configuration* d *of multi-stack pushdown system* (Q, Γ, l, δ) *is reachable from configuration* c *in* m*-phases if there are* $\alpha_1, \dots, \alpha_m$ *where each* α_i *is a sequence of* mpds *moves with pop moves from at most one stack and* $c \overset{\alpha_1}{\twoheadrightarrow} c_1 \overset{\alpha_2}{\twoheadrightarrow} c_2 \dots \overset{\alpha_m}{\twoheadrightarrow} c_m = d$.

Definition 5. *Let $\mathcal{M}$ be an mpds (Q, Γ, l, δ). Let C be a set of stack configurations of $\mathcal{M}$. We define the following.*

- $pre^*_{\mathcal{M}}(C) = \{c \mid \exists c' \in C[c \twoheadrightarrow c']\}$.
- $pre^*_{\mathcal{M}}(C, k) = \{c \mid \exists c' \in C[c' \textit{is reachable from } c \textit{ in at most } k \textit{ phases}]\}$

The set $pre^*_{\mathcal{M}}(C)$ of configurations may be uncomputable even for a fixed $\mathcal{M}$ and a fixed finite set C. Therefore we defined above $pre^*_{\mathcal{M}}(C, k)$. It is further useful to identify a subset of $pre^*_{\mathcal{M}}(C, 1)$, which consists of configurations of $\mathcal{M}$ from where an element of C may be reached by *pop* operations on stack-i and *push* operations on any stack of $\mathcal{M}$.

Definition 6. *Let $\mathcal{M}$ be an mpds (Q, Γ, l, δ). We define $c \longrightarrow_i c'$ iff there is a sequence $\overline{t}$ of transition of $\mathcal{M}$ in which pop operations occur only on stack-i and $c \xrightarrow{\overline{t}} c'$. Let C be a set of configurations of $\mathcal{M}$.*
We define $pre^*_i(C) = \{c \mid \exists c' \in C[c \longrightarrow_i c']\}$.

2.2 Regular Sets of MPDS Configurations

Definition 7. *Let $\mathcal{M}$ be an mpds (Q, Γ, l, δ) and let $c = (q, s_1, \ldots, s_l)$ be a configuration of $\mathcal{M}$. We define $\#(c) = \#_1 s_1 \#_2 s_2 \#_3 \ldots \#_l s_l \#_{l+1}$.*

Expression $\#(c)$ denotes a representation of contents of stacks in c as a string over alphabet $\Gamma^{\#} = \Gamma \cup \{\#_1, \ldots, \#_{l+1}\}$.

We define regular sets of *mpds* configurations using finite Multi-automata which were introduced in [4] for defining regular sets of configurations of (single stack) pushdown systems. Let $\mathcal{M}$ be an mpds (Q, Γ, l, δ). A finite (multi)automaton for recognizing configurations of $\mathcal{M}$ is given as $\mathcal{A} = (Q_{\mathcal{A}}, \Gamma^{\#}, Q, \delta, F)$, where $Q_{\mathcal{A}}$ is the set of states of $\mathcal{A}$, δ is the transition relation of $\mathcal{A}$, $Q \subseteq Q_{\mathcal{A}}$ is the set of its initial states and F is the set of final states of $\mathcal{A}$.

For any automaton $\mathcal{B}$ and a state q of it, we let $\mathcal{L}(\mathcal{B}, q)$ denote the set of strings accepted by $\mathcal{B}$ when started in state q. The set of configurations $C_{\mathcal{A}}$, accepted by multi-automaton $\mathcal{A}$ above is given as
$C_{\mathcal{A}} = \mathcal{L}(\mathcal{A}) = \{(q, s_1, s_2, \ldots, s_l) \mid \#_1 s_1 \#_2 s_2 \#_3 \ldots \#_l s_l \#_{l+1} \in \mathcal{L}(\mathcal{A}, q), q \in Q\}$.

Definition 8. *A set $\mathcal{C}$ of configurations of $\mathcal{M}$ is regular if there is a finite multi-automaton which accepts $\mathcal{C}$.*

In [11], a regular set C of *mpds* configurations is defined as a finite union of pairs of the form $(q, \prod_{1 \le i \le l} R_i)$, where $q \in Q$ and R_i's are regular sets over Γ. A configuration $c = (q, s_1, \ldots, s_l)$ belongs to $(q', \prod_{1 \le i \le l} R_i)$ iff $q = q'$ and for $1 \le i \le l$, $s_i \in R_i$.

It is not difficult to see that the notion of regularity in [11] and the one introduced in the definition 8 above using multi-automata are equivalent, i.e. they define the same class of sets of *mpds* configurations. This is stated in lemma below.

Lemma 1. *Let $\mathcal{M}$ be an mpds (Q, Γ, l, δ).*

1. *Let $C = \bigcup(q, \prod_{1 \le i \le l} R_i)$ be a set of configurations of $\mathcal{M}$, where $q \in Q$, R_i's are regular sets over Γ and the union is over finitely many pairs. We can design a multi- automaton $\mathcal{A}$ s.t. $\mathcal{L}(\mathcal{A}) = C$.*
2. *Given a multi- automaton $\mathcal{A}$ accepting a set of configurations of $\mathcal{M}$, we can write $\mathcal{L}(\mathcal{A})$ as a finite union of sets of the form $(q, \prod_{1 \le i \le l} R_i)$, where $q \in Q$ and each R_i is a regular set over Γ.*

Proof. An easy proof is given in the full version. □

It is possible to consider other notions of regularity for *mpds* configurations, for example by reading contents of all stacks simultaneously rather than sequentially. This will be similar to l-ary synchronized rational relations of [14], where l is the number of stacks in the *mpds*. However, in this paper we stick to the notion of regularity presented in definition 8 and work with the multi-automaton formulation only.

For a finite automaton A, we use notations $q \in \delta_A^*(p, w)$, $(p, w, q) \in \delta_A^*$ or $p \xrightarrow[A]{w} q$ to mean that on string w there is a run of A starting in state p and ending in state q.

2.3 An Example for $Post^*(C, 1)$

Let C be a set of configurations of a *mpds*. Analogous to $pre^*(C)$ and $pre^*(C, k)$, the sets $post^*(C)$ and $post^*(C, k)$ are defined based on the set of configurations that can be reached *from* elements of C.

- $post^*(C) = \{d \mid \exists c \in C[c \twoheadrightarrow d]\}$.
- $post^*(C, k) = \{d \mid \exists c \in C[d \text{ is reachable from } c \text{ in at most } k \text{ phases}]\}$

We now present an example, modified from [7], in which $post^*(C, 1)$ is not regular for a regular set C.

Let $M = (\{q_1, q_2, q_3\}, \{a, b, c\}, 3, \delta = \delta_{ins} \cup \delta_{rem})$ be an *mpds*, where $\delta_{ins} = \{(q_2, b, q_3, 2, b), (q_3, c, q_1, 3, c)\}$ and $\delta_{rem} = \{(q_1, a, q_2, 1)\}$.

Mpds M has three stacks, it pops an a from stack-1 and pushes a b on stack-2 and a c on stack-3. This is repeated till stack-1 becomes empty.

Let $C = \{(q_1, \#_1 a^m \#_2 b \#_3 c \#_4) \mid m \ge 0\}$. Clearly, C is a regular set. The set of configurations of $post^*(C)$ in control state q_1 is $D = \{(q_1, \#_1 a^* \#_2 b^m \#_3 c^m \#_4) \mid m \ge 1\}$. The set $post^*(C)$ can not be represented as any multi-automation $\mathcal{B}$ because $\mathcal{B}$ when started in state q_1 is expected to accept set $\{\#_1 a^* \#_2 b^m \#_3 c^m \#_4 \mid m \ge 0\}$, which is *not* a regular set.

Since M has only one phase (it pops from stack 1 only), $post^*(C) = post^*(C, 1)$ for M. Therefore $post^*(C, 1)$ is not regular for M.

3 Intuitive Idea for Regularity of $Pre_i^*(C)$

The main step in our proof of showing $pre^*(C, k)$ regular is to show that $pre_i^*(C)$ is regular. In this section we explain why we may expect that $pre_i^*(C)$ is regular.

In the next few sections we develop these ideas formally to construct a multi-automaton recognizing $pre_i^*(C)$.

Let $\mathcal{M} = (Q, \Gamma, l, \delta)$ be an *mpds* and let $\mathcal{A} = (Q_\mathcal{A}, \Gamma^\#, Q, \delta_\mathcal{A}, \mathcal{F})$ be a nondeterministic finite multi-automaton accepting a set $\mathcal{C}_\mathcal{A}$ of configurations of $\mathcal{M}$.

Consider a run of $\mathcal{M}$ in a phase where *pop* operation is allowed in stack-i only. In this phase contents of the other stacks change in a certain simple way. For example, if stack-j, $j \neq i$, is $\gamma_j.x_j$ at some point in this phase then later during this phase any symbol below γ_j (the topmost symbol) remains unchanged. The symbol γ_j also can't be popped but may change into another symbol because of the exchange move. A *push* instruction may push γ' on stack-j so that its contents become $\gamma'.\gamma_j.x_j$, now $\gamma_j.x_j$ can't change during the rest of this phase. So if at the beginning of the phase, stack-j, $j \neq i$ is $\gamma_j.x_j$ then at any time during this phase it will be $\alpha_j u_j x_j$, for some $\alpha_j \in \Gamma$ and $u_j \in \Gamma^*$.

We need u_j to decide if after the current phase the resulting configuration is in $\mathcal{C}_\mathcal{A}$. The string u_j is not needed in any other way as we are working for a single phase. For this purpose, the relevant information about u_j, is how automaton $\mathcal{A}$ acts on string u_j. This is captured in the pairs (q_j, q'_j) s.t. $q'_j \in \delta^*_\mathcal{A}(q_j, u_j)$. This information is *finitary*. This suggests an automaton, T_i whose states store, apart from a state of $\mathcal{M}$, also for each j, $j \neq i$, α_j and a pair (q_j, q'_j) corresponding to u_j. Automaton T_i will have transitions added corresponding to $\mathcal{M}$'s operations on stack-i in the same ways as saturation procedure does. In addition T_i will also have transitions corresponding to operations of $\mathcal{M}$ on stack-j, $j \neq i$, these transitions will update the triple (α_j, q_j, q'_j) and the current state of $\mathcal{M}$ stored in a state of T_i.

Using nondeterminism it suffices for T_i to keep in it's state for each j, $j \neq i$, only one pair (q_j, q'_j) of states for stack-j, rather than all such pairs for stack-j. Different runs of T_i may keep different pairs (q_j, q'_j) for stack-j, (even for same u_j), and on some run of T_i desired combination of pairs for all stacks will get guessed.

We describe the automaton T_i formally in the next section. In section 5, using T_i we complete the construction of automaton recognizing $pre_i^*(C_\mathcal{A})$.

4 The Automaton T_i

For $1 \leq i \leq l$, we define automaton T_i which is the main component in designing automaton to accept $pre_i^*(C)$. Let

- $S = \Gamma \times Q_\mathcal{A} \times Q_\mathcal{A}$
- $Q_{i1} = \{(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_l) \mid a_j \in S,\ for\ j \neq i,\ b \in Q\}$
- $Q_{i2} = \{(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_l) \mid a_j \in S,\ for\ j \neq i,\ b \in Q \times Q_\mathcal{A} \times Q_\mathcal{A}\}$
- $Q_i = Q_{i1} \cup Q_{i2}$

State set of T_i is Q_i. As mentioned above each a_j, $j \neq i$, keeps information (α_j, q_j, q'_j), where $q'_j \in \delta^*_\mathcal{A}(q_j, u_j)$, about contents of stack-j. T_i takes as it's input the contents of stack-i, it is designed using saturation with respect to operations of stack-i. States in Q_{i1} allow T_i to keep current state of $\mathcal{M}$ in its

i^{th} component. The states in Q_{i2} with triple (q, z_1, z_2) in their i^{th} component indicate that $\mathcal{M}$ after applying some transitions can reach a configuration c' in state q with contents of stack-i being x and $z_2 \in \delta^*_{\mathcal{A}}(z_1, x)$. The components a_j, $j \neq i$, of such a Q_{i2} state correspond to contents of stack-j in c'.

We define $T_i = (Q_i, \Gamma^{\#}, \delta_{T_i})$. The transition function δ_{T_i} is defined as $\bigcup_{j \geq 0} \delta_j$, where δ_j, $j = 0, 1, 2, \ldots$ are defined below iteratively. The sequence δ_j, $j = 0, 1, 2, \ldots$ is monotone when viewed as a relation on $Q_i \times \Gamma^{\#} \times Q_i$. This part of the construction is called saturation procedure [4]. Each triple in δ_j corresponds to a transition of $\mathcal{M}$ or $\mathcal{A}$, so we group triples in δ_j according to transition of $\mathcal{M}$ or $\mathcal{A}$, shown in boldface.

We use notation like $\overline{c} = (c_j | 1 \leq j \leq l)$, for a sequence and use $\overline{c}[x/i]$ to mean the sequence which is same as $\overline{c}$ except at index i where it is x. To keep the notation compact we write a state $(a_1, \ldots, a_{i-1}, b, a_{i+1}, \ldots, a_l)$ as $\overline{a}[b/i]$. In the following we assume that variable r ranges over integers in $[1, l] - \{i\}$, without explicitly repeating its range in each use. We let $a_r = (\gamma_r, q_r, q'_r)$ in the following transitions.

- $\delta_0 = \emptyset$, For $h \geq 0$, $\delta_{h+1} = \delta_h \cup \{\text{triples given by the following rules}\}$.

1 $(\mathbf{q}, \gamma_\mathbf{j}, \mathbf{q}', \mathbf{j}, \gamma') \in \delta_{\mathbf{exch}}$, $\mathbf{j} \neq \mathbf{i}$.
$(\overline{a}[q/i], \epsilon, \overline{a}'[q'/i]) \in \delta_{h+1}$
where $a'_j = (\gamma', q_j, q'_j)$ and $a'_m = a_m$ for $m \in [1, l] - \{j\}$.

2 $(\mathbf{q}, \gamma_\mathbf{j}, \mathbf{q}', \mathbf{j}, \gamma') \in \delta_{\mathbf{ins}}$, $\mathbf{j} \neq \mathbf{i}$.
$(\overline{a}[q/i], \epsilon, \overline{a}'[q'/i]) \in \delta_{h+1}$
where $a'_j = (\gamma', q''_j, q'_j)$, $a'_m = a_m$ for $m \in [1, l] - \{j\}$ and $(q''_j, \gamma_j, q_j) \in \delta^*_{\mathcal{A}}$.

3 $(\mathbf{q}, \gamma_\mathbf{i}, \mathbf{q}', \mathbf{i}) \in \delta_{\mathbf{rem}}$
$(\overline{a}[q/i], \gamma_i, \overline{a}[q'/i]) \in \delta_{h+1}$

4 $(\mathbf{q}, \gamma_\mathbf{i}, \mathbf{q}', \mathbf{i}, \gamma') \in \delta_{\mathbf{exch}}$
$(\overline{a}[q/i], \gamma_i, g) \in \delta_{h+1}$ where $(\overline{a}[q'/i], \gamma', g) \in \delta^*_h$

5 $(\mathbf{q}, \gamma_\mathbf{i}, \mathbf{q}', \mathbf{i}, \gamma') \in \delta_{\mathbf{ins}}$
$(\overline{a}[q/i], \gamma_i, g) \in \delta_{h+1}$ where $(\overline{a}[q'/i], \gamma'\gamma_i, g) \in \delta^*_h$

6 $(\mathbf{p}, \gamma_\mathbf{i}, \mathbf{p}') \in \delta_{\mathcal{A}}$, $\gamma_i \in \Gamma \cup \{\epsilon\}$

(a) $(\overline{a}[q/i], \epsilon, \overline{a}[(q, p_1, p_1)/i]) \in \delta_{h+1}$ for all $p_1 \in Q_{\mathcal{A}}$.

(b) $(\overline{a}[(q, p_1, p)/i], \gamma_i, \overline{a}[(q, p_1, p')/i]) \in \delta_{h+1}$ for all $p_1 \in Q_{\mathcal{A}}$.

- $\delta_{T_i} = \bigcup_{j \geq 0} \delta_j$,

The automaton T_i simulates transitions of $\mathcal{M}$ and $\mathcal{A}$ on its states. Transitions of T_i can be divided in three groups.

The first group consists of transitions (1) and (2). These transitions are for operations on stack-j, $j \neq i$. Since, T_i does not read contents of stack-j, $j \neq i$, these transitions are ϵ transitions. Let us explain transition (2). In this transition

stack-j becomes $\gamma'\gamma_j u_j$ and we have $q_j'' \xrightarrow[\mathcal{A}]{\gamma_j} q_j \xrightarrow[\mathcal{A}]{u_j} q_j'$. We use notation $\delta^*_{\mathcal{A}}$ instead of $\delta_{\mathcal{A}}$ to account for ϵ transitions in $\mathcal{A}$.

The second group consists of the transitions (3), (4) and (5). These are usual saturation operations. These relate to $\mathcal{M}$'s operations of stack-i and consume as input, a symbol of stack-i. The automaton T_i after reading a symbol from stack-i keeps track of the states which $\mathcal{M}$ can reach after this symbol has been popped. For exchange and push on stack-i this is kept track of by new transitions added iteratively, by saturation procedure. Note that the transitions added iteratively include the influence of transitions in *all* groups.

The third group consists of transitions (6a) and (6b). The automaton T_i uses an approach similar to that in [12], at any point (that is after any number of transitions of $\mathcal{M}$), T_i may choose to simulate $\mathcal{A}$ on contents of stack-i. Transition (6a) starts simulating $\mathcal{A}$ from state p on contents of stack-i in the current configuration. Transition (6b) continues the simulation of $\mathcal{A}$ in the last component of triple (q, p_1, p). Note that after applying a transition from this group, T_i can not apply transitions of any other group. Transitions of this group also participate in saturation procedure.

Note that only transitions in step (4), (5) depend on transitions present in the previous stage. Other transitions are in δ_h for all $h > 0$.

T_i does not have any initial or final state. It's purpose is not to recognize any language instead it is used as a component in the automaton to recognize $pre^*_i(\mathcal{C}_{\mathcal{A}})$. In the next section we prove some crucial properties of automaton T_i.

4.1 Properties of T_i

States of T_i abstract configurations of c. The relations between the two is described in the definitions below.

Definition 9. *Let $c = (q, \overline{\gamma v})$ be an mpds configuration, where $\overline{\gamma v} = \gamma_1 v_1, \ldots, \gamma_l v_l$. Let $e = \overline{a}[q/i]$ be a state of T_i, where $a_r = (\gamma_r, q_r, q_r')$ for $r \in [1, l] - i$. We define $c \approx e$ (read as c is compatible with e) if $q_r' \in \delta^*_{\mathcal{A}}(q_r, v_r)$ for $r \in [1, l] - i$.*

Definition 10. *Let $c = (q, \overline{\gamma v}[w/i])$ be an mpds configuration, where $\overline{\gamma v} = \gamma_1 v_1, \ldots, \gamma_l v_l$. Let $e = \overline{a}[(q, z_1, z_2)/i]$ be a state of T_i. We define $c \approx_1 e$ (read as c is compatible with e) if $c \approx e$ and $z_2 \in \delta^*_{\mathcal{A}}(z_1, w)$.*

In relationship $c \approx e$ contents of stacks $[1, l] - \{i\}$ are abstracted by a run of $\mathcal{A}$. As for a given $v \in \Gamma^*$, there can be many pairs (q, q') s.t. $q' \in \delta^*_{\mathcal{A}}(q, v)$, there are many e for a given c s.t. $c \approx e$. Relation $c \approx e$ is independent of contents of stack-i. In $c \approx_1 e$ contents of stack-i are also abstracted by a run of $\mathcal{A}$.

The lemma below shows that transition rules of T_i capture the effect of transitions of $\mathcal{M}$ and transitions of $\mathcal{A}$ on actual configurations, in sound and complete way, on abstracted states of T_i.

Lemma 2. *Let $c = (q, \overline{\gamma v}[w/i])$ be an mpds configuration. Then the following hold.*

1. *If* $c \longrightarrow_i c'$, $c' \approx_1 e'$ *then there is an* e, $c \approx e$ *and* $(e, w, e') \in \delta^*_{T_i}$.
2. *If* $c \approx e$, $(e, w, e') \in \delta^*_{T_i}$ *where* $e' \in Q_{i2}$ *then there is a* c' *s.t.* $c \longrightarrow_i c'$ *and* $c' \approx_1 e'$.

Proof. The proof is a bit tedious but works out as expected. We use induction and case analysis in same way as in saturation proofs of [12]. Details are in full version. □

5 Regularity of $Pre^*_i(C_{\mathcal{A}})$

In this section we construct an automaton P_i, using T_i of the previous section, to accept $pre^*_i(C_{\mathcal{A}})$.

Let $\mathcal{A}$ be a multi-automaton, $\overline{\gamma} = \gamma_1 \dots \gamma_l$ and $e = \overline{a}[(q, z_1, z_2)/i]$, where $a_r = (\alpha_r, p_r, q_r)$. We know that e encodes information related to *some run* of $\mathcal{A}$, on stack contents modified during phase-i (where *pop* on stack-i only is allowed). Definition 11 below, refines the acceptance by $\mathcal{A}$ to acceptance via a run which is consistent with the information in e.

First some notation, given a run ρ of $\mathcal{A}$ on input y, if x has unique occurrence in y then we let $\rho(x)$ be the state reached in run ρ at the end of x.

Definition 11. *Let* $(q, \overline{\alpha u}[w'/i]) \approx_1 e$. *We say that* $(q, \overline{\alpha u v}[w'/i]) \in \mathcal{L}(\mathcal{A})$ ***via*** e *if there is an accepting run* ρ *of* $\mathcal{A}$ *on input* $\#(q, \overline{\alpha u v}[w'/i])$, *starting at state* q *s.t.* $\rho(\#_r \alpha_r) = p_r$, $\rho(\#_r \alpha_r u_r) = q_r$, *for* $r \neq i$, *and* $\rho(\#_i) = z_1$, $\rho(\#_i w') = z_2$.

The definition above can be seen as stating that a configuration is accepted by $\mathcal{A}$ using a run that is in some specified states at some specific points in the input.

Let e be as above, we wish to design an automaton which accepts a configuration c such that if in phase-i, c is transformed according to information in e then the resulting configuration is accepted by $\mathcal{A}$ via e. We call this multi-automaton $\mathcal{A}^{\overline{\gamma}}_e$ which works as follows. Automaton $\mathcal{A}^{\overline{\gamma}}_e$ on input $\#(p, \overline{\gamma v}[w/i])$, simulates $\mathcal{A}$ from start state q with the following modifications.

On reading $\#_j \gamma_j$, $j \neq i$, $\mathcal{A}^{\overline{\gamma}}_e$ simulates $\mathcal{A}$ on $\#_j \alpha_j$ instead of on $\#_j \gamma_j$. If after having been simulated on $\#_j \alpha_j$ state of $\mathcal{A}$ is not p_j then $\mathcal{A}^{\overline{\gamma}}_e$ aborts otherwise $\mathcal{A}^{\overline{\gamma}}_e$ changes the state of $\mathcal{A}$ in simulation to q_j and continues the simulation of $\mathcal{A}$ on the input following $\#_j \gamma_j$. (This ensures that on $\#_j \gamma_j$, $\mathcal{A}^{\overline{\gamma}}_e$ simulates $\mathcal{A}$ on all $\#_j \alpha_j u_j$, s.t. $p_j \xrightarrow[\mathcal{A}]{u_j} q_j$.)

On reading $\#_i \gamma_i$, if state of the simulated $\mathcal{A}$ is not z_1 then $\mathcal{A}^{\overline{\gamma}}_e$ aborts otherwise $\mathcal{A}^{\overline{\gamma}}_e$ changes the state of $\mathcal{A}$ to z_2 and $\mathcal{A}$ ignores the input till it sees $\#_{i+1}$. (This ensures that on $\#_i w \#_{i+1}$, $\mathcal{A}^{\overline{\gamma}}_e$ simulates $\mathcal{A}$ on all $\#_i w' \#_{i+1}$, where $z_1 \xrightarrow[\mathcal{A}]{w'} z_2$.)

Finally, $\mathcal{A}^{\overline{\gamma}}_e$ accepts if it reaches accepting state of $\mathcal{A}$ in the simulation at the end of the given input.

The following Lemma is clear from the construction of $\mathcal{A}^{\overline{\gamma}}_e$.

Lemma 3. *For any* $(q, \overline{\alpha u}[w'/i]) \approx_1 e$, $(p, \overline{\gamma v}[w/i]) \in \mathcal{L}(\mathcal{A}^{\overline{\gamma}}_e)$ *iff* $(q, \overline{\alpha u v}[w'/i]) \in \mathcal{L}(\mathcal{A})$ *via* e.

Following lemma justifies the construction of $\mathcal{A}_e^{\overline{\gamma}}$ as it gives a criteria for a configuration c to be in $pre_i^*(\mathcal{C}_\mathcal{A})$ using $\mathcal{A}_e^{\overline{\gamma}}$.

Lemma 4. *Let $\mathcal{A}$ be a multi-automaton accepting a set $\mathcal{C}_\mathcal{A}$ of configurations of $\mathcal{M}$. Then $c = (p, \overline{\gamma v}[w/i]) \in pre_i^*(\mathcal{C}_\mathcal{A})$ iff there are $e \in Q_{i1}$ and $e' \in Q_{i2}$ s.t. $(p, \overline{\gamma}[w/i]) \approx e$, $(e, w, e') \in \delta_{T_i}^*$ and $c \in L(\mathcal{A}_{e'}^{\overline{\gamma}})$.*

Proof. Proof is given in full version of this paper. □

Using the characterization in Lemma 4, we design a multi-automaton P_i to accept $pre_i^*(C)$ in the following lemma.

Lemma 5. *Let $\mathcal{A}$ be a multi-automaton accepting a set $\mathcal{C}_\mathcal{A}$ of configurations of $\mathcal{M}$. There is a multi-automaton P_i which accepts $pre_i^*(C_\mathcal{A})$.*

Proof. We design P_i to accept $pre_i^*(C)$ using the criteria in Lemma 4. Automaton P_i on input $c = (q, \overline{\gamma v}[w/i])$, non-deterministically guesses $e \in Q_{i1}$ and $e' \in Q_{i2}$ and separately verifies each of the three conditions (i) $(q, \overline{\gamma}[w/i]) \approx e$ (ii) $(e, w, e') \in \delta_{T_i}^*$ (iii) $c \in L(\mathcal{A}_{e'}^{\overline{\gamma}})$ as it scans the input.

In more detail, P_i when started from state q, guesses $e = \overline{a}[q/i]$, where $a_r = (\beta_r, p_r, q_r)$, $p_r \xrightarrow[\mathcal{A}]{\epsilon} q_r$ for $r \neq i$. This is hardwired in P_i as guessing of e and e' is hardwired in P_i. Condition (i) is now verified by an automaton which checks that for each $r \neq i$ the symbol just after $\#_r$ is β_r. Condition (ii) is verified by running automaton T_i from state e on the string enclosed between $\#_i$ and $\#_{i+1}$ and accepting if it reaches e'. Condition (iii) is verified by running automaton $\mathcal{A}_{e'}^{\overline{\gamma}}$, from state q on $\#c$.

The verification phase consists of running these three automata simultaneously and accepting if each of them accepts. □

6 Regularity of $Pre^*(C, k)$

From the previous section, for each i, $1 \leq i \leq l$, we have multi-automaton $P_i(\mathcal{A}) = (Q_i, \Gamma^\#, \delta_i, Q, F_i)$ which accepts $pre_i^*(\mathcal{C}_\mathcal{A})$. Consider a multi-automaton $P(\mathcal{A}) = ((\oplus_{i=1}^l Q_i) \oplus Q, \Gamma^\#, (\oplus_{i=1}^l \delta_i) \cup \{(q, \epsilon, q^i) | q \in Q\}, Q, \cup_{i=1}^l F_i)$, where symbol $\oplus$ stands for a disjoint union.

$P(\mathcal{A})$ is obtained by adding a new copy of states Q as initial states of $P(\mathcal{A})$ and ϵ transitions from each initial state $q \in Q$ to corresponding initial state of each multi-automaton $P_i(\mathcal{A})$. The other transitions in $P(\mathcal{A})$ are the transitions present in each $P_i(\mathcal{A})$.

In state $q \in Q$, on input $\#(c)$, $P(\mathcal{A})$ nondeterministically chooses i and simulates $P_i(\mathcal{A})$ on $\#(c)$. Thus the language accepted by $P(\mathcal{A})$ is $\bigcup_i pre_i^*(\mathcal{C}_\mathcal{A})$. That is $P(\mathcal{A})$ accepts the set of configurations of $\mathcal{M}$ from which $\mathcal{M}$ can reach an element of $\mathcal{C}_\mathcal{A}$ in a single phase.

Finally, we define $P^0 = \mathcal{A}$ and $P^{j+1}(\mathcal{A}) = P(P^j(\mathcal{A}))$, for $j \geq 0$. The automaton $P^k(\mathcal{A})$, which is defined by iterating the operator P, k-times, accepts the set of configurations of $\mathcal{M}$ from which $\mathcal{M}$ can reach an element of $\mathcal{C}_\mathcal{A}$ in at most k phases. That is $P^k(\mathcal{A})$ accepts $pre^*(C_\mathcal{A}, k)$.

6.1 Complexity

Let the number of states in multi-automaton $\mathcal{A}$ be $|Q_{\mathcal{A}}|$ and the number of states in *mpds* $\mathcal{M}$ be $|Q|$. It is clear that $|Q| \leq |Q_{\mathcal{A}}|$, as $Q \subseteq Q_{\mathcal{A}}$. A simple counting shows the following.

Number of states in T_i is $O(|\Gamma|^l.|Q_{\mathcal{A}}|^{2l+1})$.

Number of states in $\mathcal{A}^{\overline{\gamma}}_{e'}$ is $O(|Q_{\mathcal{A}}|)$.

Number of states in $P_i(\mathcal{A})$ is $O((|\Gamma|^l \cdot |Q_{\mathcal{A}}|^{2l+1})^2 \cdot (|\Gamma|^l \cdot |Q_{\mathcal{A}}|^{2l+1} \cdot |Q_{\mathcal{A}}|)) = O(|\Gamma|^{3l} \cdot |Q_{\mathcal{A}}|^{7l})$.

(The number of different e, e' pairs is $O((|\Gamma|^l.|Q_{\mathcal{A}}|^{2l+1})^2)$. For each guess of e, e' pair, $P_i(\mathcal{A})$ has a copy of T_i and $\mathcal{A}^{\overline{\gamma}}_{e'}$ to simulate these automata on the input.)

Number of states in $P(\mathcal{A})$ is $O(l.|\Gamma|^{3l}.|Q_{\mathcal{A}}|^{7l})$.

This can be taken as $O(|Q_{\mathcal{A}}|^{c.l})$, for some constant c, if $|\Gamma|$ is taken as constant or $|Q_{\mathcal{A}}| \geq |\Gamma|$.

The number of states in $P^k(\mathcal{A})$ is therefore $O(|Q_{\mathcal{A}}|^{(c.l)^k})$.

We sum all this up in the main theorem below.

Theorem 1. *Let $\mathcal{A}$ be a multi-automaton with $|Q_{\mathcal{A}}|$ many states recognizing a set $C_{\mathcal{A}}$ of configurations of a l stack mpds $\mathcal{M}$. There is a multi-automaton with $O(|Q_{\mathcal{A}}|^{(c.l)^k})$ states (where c is a constant independent of $\mathcal{A}$ and $\mathcal{M}$), which recognizes $pre^*_{\mathcal{M}}(C_{\mathcal{A}}, k)$ and can be constructed in $O(|Q_{\mathcal{A}}|^{(c.l)^k})$ time.*

Note that the automaton constructed permits incremental construction as the number of phases is increased. Suppose we have the automaton $P^k(\mathcal{A})$ to recognize $pre^*(C_{\mathcal{A}}, k)$, now to construct the automaton $P^{k+1}(\mathcal{A})$ to recognize $pre^*(C_{\mathcal{A}}, k+1)$, we do not need to the start the construction from scratch instead we may just apply operator P to $P^k(\mathcal{A})$. This feature may be useful in practice.

7 Extracting a Witness Sequence of Transitions

Given an accepting run R of $P^k(\mathcal{A})$ on $c = (q, \overline{\gamma v})$, we show how to effectively construct a sequence $\overline{t}$ of transitions of $\mathcal{M}$, s.t. $c \xrightarrow{\overline{t}} c'$ for a $c' \in \mathcal{A}$.

We begin by observing that Lemma 2 part (2) and Lemma 3 are constructive in the following sense.

Lemma 6. *Let $c = (q, \overline{\gamma v})$ be an mpds configuration and let $e = \overline{a}[q/i] \in Q_{i1}$ where $a_r = (\gamma_r, p_r, q_r)$. Let $c \approx e$. Further a run $e \xrightarrow[T_i(\mathcal{A})]{\gamma_i v_i} e'$ of $T_i(\mathcal{A})$ be given for some $e' = \overline{a'}[(q', z_1, z_2)/i] \in Q_{i2}$, where $a'_r = (\alpha_r, p'_r, q'_r)$.*

Then we can effectively construct $\overline{t}$ and c' s.t. $c \xrightarrow{\overline{t}}_i c' = (q', \overline{\alpha u v})[w/i]$. Further, we can effectively construct runs $p'_r \xrightarrow[\mathcal{A}]{u_r v_r} q'_r$ and $z_1 \xrightarrow[\mathcal{A}]{w} z_2$.

Proof. Easily follows from the proof of Lemma 2, part (2). Details are given in full version. □

Following lemma is immediate from the construction of $\mathcal{A}^{\gamma}_{e'}$ in section 5. Instead of $\mathcal{A}$, we consider an arbitrary multi-automaton $\mathcal{B}$ accepting configurations of a *mpds*.

Lemma 7. *Let* $e' = \overline{a'}[(q, z_1, z_2)/i] \in Q_{i2}$ *where* $a'_r = (\alpha_r, p'_r, q'_r)$. *Also let* $(q, \overline{\alpha u}[w/i]) \approx_1 e'$. *From an accepting run of* $\mathcal{B}^{\gamma}_{e'}$ *on* $c = (p, \overline{\gamma v})$ *and runs* $p'_r \xrightarrow[\mathcal{B}]{u_r} q'_r$ *and* $z_1 \xrightarrow[\mathcal{B}]{w} z_2$ *we can easily construct an accepting run of* $\mathcal{B}$ *on* $(q, \overline{\alpha u v})[w/i])$.

Proof. Let an accepting run of $\mathcal{B}^{\gamma}_{e'}$ on $\#(p, \overline{\gamma v})$ be given. From this we can extract a run R of $\mathcal{B}$ on c with modifications introduced by $\mathcal{B}^{\gamma}_{e'}$. In the transitions between p'_r and q'_r in R, we insert the run $p'_r \xrightarrow[\mathcal{B}]{u_r} q'_r$ and in transition between z_1 and z_2 in R, we insert the run $z_1 \xrightarrow[\mathcal{B}]{w} z_2$. This gives an accepting run of $\mathcal{B}$ on $\#((q, \overline{\alpha u v})[w/i])$. □

Equipped with the two Lemma above, we can extract the witness run as follows. Let an accepting run R of $P^k(\mathcal{A})$ on $c = (p, \overline{\gamma y})$ be given. By definition of $P^k(\mathcal{A})$, we have a run R of $P(B)$ on $c = (p, \overline{\gamma y})$, where $B = P^{k-1}(\mathcal{A})$. By examining the first transition in R, we get an i s.t. there is an accepting run R_i of $P_i(B)$ on c. By examining R_i we get

1. $e = \overline{a}[p/i]$, where $a_r = (\gamma_r, p_r, q_r)$ and $p_r \xrightarrow[B]{\epsilon} q_r$ for $r \neq i$.
2. $e' = \overline{a'}[(q, z_1, z_2)/i]$, where $a'_r = (\alpha_r, p'_r, q'_r)$.
3. Run $e \xrightarrow[T_i(B)]{\gamma_i y_i} e'$ of $T_i(B)$.
4. Accepting run of $B^{\gamma}_{e'}$ on $c = (p, \overline{\gamma y})$.

By Lemma 6 (taking $v_r = \epsilon$), we can effectively get $\overline{t_k}$, c'_k s.t. $(p, \overline{\gamma}) \xrightarrow{\overline{t_k}}_i c'_k = (q, \overline{\alpha u})[w/i]$ and $(q, \overline{\alpha u}[w/i]) \approx_1 e'$, and also runs $p'_r \xrightarrow[B]{u_r} q'_r$ and $z_1 \xrightarrow[B]{w} z_2$. It follows that $c \xrightarrow{\overline{t_k}}_i c_k = (q, \overline{\alpha u y})[w/i]$. As $c = (p, \overline{\gamma y}) \in L(B^{\overline{\gamma}}_{e'})$, by Lemma 7, we can effectively get an accepting run of B on $c_k = (q, \overline{\alpha u y}[w/i])$. As $c \xrightarrow{\overline{t_k}}_i c_k$ and $B = P^{k-1}(\mathcal{A})$ has an accepting run on c_k, we can repeat the reasoning to get $c_k \xrightarrow{\overline{t_{k-1}}}_{i'} c_{k-1}$, for some i' and an accepting run of $P^{k-2}(\mathcal{A})$ on c_{k-1}. Continuing the process, finally we get t_1 s.t. $c_2 \xrightarrow{\overline{t_1}}_j c_1$, for some j, and $c_1 \in \mathcal{A}$. The desired witness sequence is $\overline{t} = \overline{t_k}, \ldots, \overline{t_1}$. This gives us the theorem below.

Theorem 2. *Let* $\mathcal{A}$, $C_{\mathcal{A}}$ *and* $\mathcal{M}$ *be as in theorem 1. Let* $P^k(\mathcal{A})$ *be the multi-automaton constructed in the proof of theorem 1 to recognize* $pre^*_{\mathcal{M}}(C_{\mathcal{A}}, k)$. *Given an accepting run* R *of* $P^k(\mathcal{A})$ *on configuration* d, *a sequence* $\overline{t}$ *of transitions of* $\mathcal{M}$ *can be effectively constructed, s.t.* $d \xrightarrow[\mathcal{M}]{\overline{t}} d'$ *for some* $d' \in \mathcal{A}$.

Preliminary analysis of complexity of the above procedure for extracting a witness sequence, indicates that it is linear in length of the run R but triple exponential in the number of phases for fixed $\mathcal{M}$ and $\mathcal{A}$.

8 Conclusion

We have shown that the set $pre^*(C, k)$, for a k phase bounded *mpds* is regular and an automaton representation of this set can be constructed efficiently. We have also given a procedure to extract from any accepting run of this automaton on a configuration d, a witness sequence of transitions of *mpds* to show that $d \in pre^*(C, k)$. Note that these results also apply to systems which can be simulated by phase bounded *mpds*. As an example, these results apply of message passing systems in [8]. Representation of configurations of these systems involves representing contents of message queues between processes, each such queue can be represented by two stacks as in the simulation of [8].

We note that the construction of pre^* also gives a new proof for checking emptiness of a bounded phase *mpda*. Let $M = (Q, \Sigma, \Gamma, l, q_0, \delta, q_f)$ be a l stack *mpda*, where q_0 is the initial state, acceptance is by final state q_f and other symbols have the usual meaning. We obtain a *mpds* $M1$ from M by erasing input symbols from transitions in δ. Let $L(M, k)$ be the language of words recognized by M in k-phases. Now, $L(M, k) = \emptyset$ iff $I \in pre^*_{M1}(C_f, k)$, where $C_f = (q_f, \#_1\Gamma^*\#_2 \ldots \#_l\Gamma^*\#_{l+1})$ is the set of final configurations of M and $I = (q_0, \#_1\#_2 \ldots \#_l\#_{l+1})$ is the initial configuration M. Therefore emptiness of $L(M, k)$ can be determined by checking membership of I in finite multi-automaton for $pre^*_{M1}(C_f, k)$. This algorithm is very different from the other known methods [7,13], for checking emptiness of a *mpda*.

It may be interesting to see if results similar to ours can also be shown for a little more general model, the multi-pushdown systems of [1], where stacks are ordered and a pop operation is always on the first non-empty stack. There is no bound on the number of phases in that system.

Another natural question is to extend these results to the setting of two players. The present approach, if extended successfully, will give a effective history free strategy in two player reachability games over bounded phase *mpds* whereas techniques of [13] can give only a strategy computable by a multi-stack pushdown automaton.

Recently, a saturation based proof has been given in [6] for constructing winning regions in parity games over a single stack pushdown system. It may be natural to see if method of [6] combined with ideas of this paper can give a saturation based construction for representation of winning regions in parity games over bounded phase *mpds*.

While the worst case complexity of our construction is optimal, it may be possible to improve this construction so that it works more efficiently on some inputs. In this direction, it will be interesting to see if automata constructions of sections 4,5 and 6 can be combined into a single construction where we start with a small number of states and new states are added only whenever they are needed.

Another way to control size of the automaton constructed may be to consider approximations of bounded phase reachability. The construction for a single phase is single exponential. It may be combined with usual bounded context switching construction. That is, for the first few iterations we do the phase switching construction and then on the automaton so obtained we do the usual

construction for bounded context switching. This will keep the double exponential growth in check. For improving the complexity in a single phase we may restrict push in a single phase to some small number of stacks rather than allowing it on all stacks. Size of the automaton constructed for $pre_i^*(C)$ is exponential in the number of stacks on which push operations are allowed in phase-i.

Acknowledgments

Financial support for this work was provided by Research I Foundation.

References

1. Atig, M.F., Bollig, B., Habermehl, P.: Emptiness of Multi-pushdown Automata Is 2ETIME-Complete. In: Ito, M., Toyama, M. (eds.) DLT 2008. LNCS, vol. 5257, pp. 121–133. Springer, Heidelberg (2008)
2. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-Bounded Analysis for Concurrent Programs with Dynamic Creation of Threads. In: TACAS 2009. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
3. Lal, A., Reps, T.W.: Reducing Concurrent Analysis under a Context Bound to Sequential Analysis. Formal Methods in System Design 35(1), 73–97 (2009)
4. Bouajjani, A., Esparza, J., Maler, O.: Reachability analysis of pushdown automata: Applications to model checking. In: Mazurkiewicz, A., Winkowski, J. (eds.) CONCUR 1997. LNCS, vol. 1243, pp. 135–150. Springer, Heidelberg (1997)
5. Hague, M., Ong, C.-H.L.: Symbolic Backwards-Reachability Analysis for Higher-Order Pushdown Systems. Logical Methods in Computer Science 4 (2008)
6. Hague, M., Ong, C.-H.L.: Winning Regions of Pushdown Parity Games: A Saturation Method. In: Bravetti, M., Zavattaro, G. (eds.) CONCUR 2009. LNCS, vol. 5710, pp. 384–398. Springer, Heidelberg (2009)
7. Madhusudan, P., Parlato, G., La Torre, S.: A Robust Class of Context-Sensitive Languages. In: Proc: LICS 2007, pp. 161–170. IEEE Computer Society, Los Alamitos (2007)
8. Madhusudan, P., Parlato, G., La Torre, S.: Context-Bounded Analysis of Concurrent Queue Systems. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 299–314. Springer, Heidelberg (2008)
9. Madhusudan, P., Parlato, G., La Torre, S.: An Infinite Automaton Characterization of Double Exponential Time. In: Kaminski, M., Martini, S. (eds.) CSL 2008. LNCS, vol. 5213, pp. 33–48. Springer, Heidelberg (2008)
10. La Torre, S., Madhusudan, P., Parlato, G.: Reducing Context-Bounded Concurrent Reachability to Sequential Reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
11. 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)
12. Seth, A.: An Alternative Construction in Symbolic Reachability Analysis of Second Order Pushdown Systems. Int. J. Found. Comput. Sci. (IJFCS) 19(4), 983–998 (2008)
13. Seth, A.: Games on Multi-Stack Pushdown Systems. In: Artemov, S., Nerode, A. (eds.) LFCS 2009. LNCS, vol. 5407, pp. 395–408. Springer, Heidelberg (2008)
14. Thomas, W.: A short introduction to infinite automata. In: Kuich, W., Rozenberg, G., Salomaa, A. (eds.) DLT 2001. LNCS, vol. 2295, pp. 130–144. Springer, Heidelberg (2002)

Model-Checking Parameterized Concurrent Programs Using Linear Interfaces*

Salvatore La Torre[1], P. Madhusudan[2], and Gennaro Parlato[2]

[1] Università degli Studi di Salerno, Italy
[2] University of Illinois at Urbana-Champaign, USA

Abstract. We consider the verification of parameterized Boolean programs— abstractions of shared-memory concurrent programs with an unbounded number of threads. We propose that such programs can be model-checked by iteratively considering the program under k-round schedules, for increasing values of k, using a novel compositional construct called *linear interfaces* that summarize the effect of a block of threads in a k-round schedule. We also develop a game-theoretic sound technique to show that k rounds of schedule suffice to explore the entire search-space, which allows us to prove a parameterized program entirely correct. We implement a symbolic model-checker, and report on experiments verifying parameterized predicate abstractions of Linux device drivers interacting with a kernel to show the efficacy of our technique.

We dedicate this paper to the memory of Amir Pnueli.

1 Introduction

Parameterized concurrent programs are concurrent programs with an *unbounded number of threads*, executing similar code (or code chosen from a finite set of programs). In the model-checking literature, parameterized programs have been heavily investigated (see section of related work), as they are a natural extension of concurrent systems, and a very relevant model for communication protocols and distributed systems. Model-checking parameterized programs, even when the data domain is finite, is, in general, *undecidable*.

In this paper, we propose a new technique to verify parameterized finite-data-domain programs, or parameterized Boolean programs. The primary idea is to *iterate* over k-round schedules of the parameterized program, for increasing values of k, and detect termination by proving that all reachable configurations have been reached at the k-th round, for some k.

More precisely, we work through phases, each phase for an increasing value of k, and model-check if the parameterized program can reach the error state, for some instantiation of n threads and in some k-round schedule. A k-round schedules consists of k rounds, where in each round every thread gets scheduled

* This work was partially funded by the MIUR grants FARB 2008-2009 Università degli Studi di Salerno (Italy), NSF CAREER award #0747041, and NSF Award #0917229.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 629–644, 2010.

once in some fixed order, and where each thread gets scheduled for an *arbitrary* number of events. This task, though an under-approximation of the reachable state-space, is challenging, as the number of threads is not fixed. We develop a novel construct, called *linear interfaces*, that summarizes the effect of an *arbitrary* block of threads in a k-round schedule. Linear-interfaces (as opposed to general interfaces), capture the effect of a block of threads along a *single run* that context-switches into and out of the block.

The lack of branching information and the finite description of linear interfaces helps us to build a compositional framework to search the state space, that combines linear interfaces without blow-up. We develop a fairly intricate algorithm that uses linear interfaces for blocks of threads scheduled at the right end of each round (*right-blocks*), to ensure that we never leave the set of reachable states in constructing linear interfaces. Further, the algorithm can be captured as a fixed-point computation over an appropriate signature, and hence naturally yields to symbolic BDD-based methods.

Our second contribution is an *adequacy check* that tries to prove that all reachable states of a parameterized program are already reached under some k-round schedule. This check, which is sound but not complete, is formulated as a two-player reachability game on an (implicitly defined) graph. Intuitively, Eve (player 0) aims to show that there is a global state reachable in the $(k+1)$-th round that is not reachable in the k-th round, and Adam (player 1) aims to disprove this. The game works by Eve *declaring* a global state, by declaring one at a time the local states on each thread, and Adam responds by reaching the same states using only k rounds. If Adam has a *winning strategy* (and hence Eve has none), then this proves that every global state reachable in the $(k+1)$-th round is already reachable in the k-th round. Thus, we can stop computing for higher values of k and declare the program correct. The idea of formulating the check as a *game* is a technical novelty, and is used to declare a state that involves an *arbitrary large* number of threads step by step (she cannot very well declare the global state in one stroke as then the game-graph will no longer be finite). However, the fact that Eve declares the global state one thread at a time can give her an advantage in the game, and if Eve has a winning strategy, we cannot conclude that a configuration is reachable in the $(k+1)$-th round and not in the k-th round. Hence our adequacy check is sound but incomplete. The game, and finding whether Adam has a winning strategy (i.e. solving the game), can also be formulated and computed symbolically.

The idea of slicing the reachable state-space in terms of the number of rounds is non-traditional (classic approaches would induct over the number of threads) and is motivated by recent work on slicing the state-spaces of concurrent programs using a bounded number of context switches. Bounded context-switching is motivated by the belief that most errors (and, in fact, most reachable states) will be already reachable in a few number of rounds [21]. Also, from an algorithmic perspective, model-checking under k-round schedules is *decidable* and can be achieved using, at any point, only *one* copy of the local state of a thread, and $O(k)$ copies of the shared variables.

Our work argues that the above can be exploited also for *parameterized* systems, thus obtaining an effective decidable way of exploring search spaces. Moreover, our *adequacy check*, which is entirely novel, can verify (soundly) that searching beyond k-round schedules is useless, and hence terminate the search, proving the parameterized program correct for any number of threads and any schedule. We emphasize that the completeness check closely follows and relies on the bounded-round schedule reachability algorithm.

While several approaches in the literature have explored bounded context-switching as an *under-approximation* to find errors, to our knowledge ours is the first to use this under-approximation to prove that the program is in fact entirely correct. Our adequacy check works for parameterized programs, but no similar check is known even for concurrent programs with finitely many threads. We thus believe that the analysis of such programs would benefit from using it.

We report on a symbolic BDD-based implementation of both the k-round model-checking for parameterized programs as well as the k-round adequacy check. Our implementation is a succinct formulation of the algorithms using fixed-points, and we use the GETAFIX framework [13] that we have developed recently, to implement our algorithm by simply writing fixed-point equations.

We report on using our model-checker to verify a large suite of Boolean parameterized programs obtained from the DDVERIFY tool, that extracts Boolean models of Linux device drivers and the OS kernel, using predicate abstraction, in order to check them against rules of kernel API usage (similar to SLAM, which is for Windows drivers). Our parameterized setting models an arbitrary number of these drivers working with the OS. We report on experiments performed on about 8000 programs and properties, and show that our tool can effectively find reachable error-states, and furthermore *prove* that more than 80% of them are entirely correct, using the adequacy check.

In summary, our theoretical and experimental results suggest a new technique for verifying parameterized programs: to effectively under-approximate them using a few round schedules (but with arbitrary number of threads), summarized and analyzed using linear interfaces, and build effective techniques to prove a few rounds suffice to reach the entire reachable state-space.

Due to lack of space, detailed proofs are omitted in the paper, but can be found in [15]. Moreover, details of the implementation of the idea presented in this paper is at the GETAFIX website: `http://www.cs.uiuc.edu/~madhu/getafix`.

Related work. Compositional verification using interfaces for modules has been investigated before: e.g. the work in [4] computes interfaces for modules using *learning* for compositional verification. However, these interfaces are modeled as *finite transition systems*, and will not help in verifying unboundedly many threads as the interfaces, when composed, will keep increasing in size.

The idea of exploring search-spaces of concurrent programs with finitely many threads, using a small number of context-switches for finding bugs has been well studied recently [21,18,20,22,17,13,14]. The CHESS tool from Microsoft espouses this philosophy by testing concurrent programs by systematically choosing schedules with a small number of context-switches/pre-emptions.

A recent paper [1] proposes a (theoretical) solution to the model-checking problem of reachability in concurrent programs with dynamic creation of threads, where a thread is context-switched into only a bounded number of times. This dynamic thread creation can model the unboundedly many threads in our setting. However, dynamic thread creation requires keeping track of the *number of threads that are in a local state*, even under bounded switching. The paper in fact shows reductions between this reachability problem and Petri-net coverability, establishing EXPSPACE-hardness. In contrast, it follows from our fixed-point formulation that the model-checking problem in our setting is PSPACE-complete. More importantly, our fixed-point formulation actually yields a practical *symbolic BDD-based solution*, while it is not clear how to build a symbolic model-checker using the Petri-net reduction given in [1] (the paper does not report any implementation or experiments).

There is a rich history of verifying parameterized asynchronously communicating concurrent programs, especially motivated by the verification of distributed protocols: sample research includes network invariants (see [12] and references therein) and its abstractions [3,10,6]; *regular model-checking* [11], using small-model theorems [7]; *split invariants* followed by abstractions based on this invariant and model-checking [5]. Symmetry in replicated concurrent processes [8] has been exploited in the Murφ tool [10].

Approaches for verifying several replicated components (though finite) have used *counter abstraction* [19], and recent work has used counter abstraction combined with cartesian representations of local and global state in order to verify a fixed number of Linux device drivers working in parallel [2]. The model-checking work we report in this paper handles the same device drivers but with an *unbounded number* of them working in parallel and restricted to a bounded number of round schedules.

Abstraction for parameterized systems have also been investigated: using predicate abstraction [16] as well as abstract-interpretation over standard abstraction domains [9].

2 Parameterized Boolean programs

We are interested in concurrent programs composed of several concurrent processes, each executing on possibly unboundedly many threads, with variables ranging only over the Boolean domain (*parameterized programs*). All threads run in parallel and share a fixed number of variables.

Each parameterized program consists of a sequential block of statements `init`, where the shared variables are initialized, and a list of concurrent processes. Each *process* is essentially a sequential program (namely, a Boolean program) with explicit syntax for nondeterminism and (recursive) function calls, along with the possibility of declaring sets of statements to be executed *atomically*. Functions are all call-by-value. Variables can be scoped locally to a function, globally to a process in a thread or shared amongst all processes in all threads. The statements in a parameterized program can refer to all variables in scope.

A parameterized program is initialized with an arbitrary finite number of threads, each thread running a copy of one process. Dynamic creation of threads is not allowed, but it can be modeled by having the threads in a "dormant" state until a message from the parent thread is received.[1]

An *execution* of a parameterized program is obtained by interleaving the behaviors of the threads which are involved in it. For a concurrent process we assume the standard semantics of sequential programs (the request of executing atomically a block of statements has no meaning when executing a single thread). Formally, let $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^{n})$ be a *parameterized program* where S is the set of shared variables and P_i is a process, $i \in [1, n]$. We assume that each statement of the program has a unique *program counter* labeling it. A *thread* T of $\mathcal{P}$ is a copy (instance) of some P_i, $i \in [1, n]$. At any point, only one thread is *active*. For any $m > 0$, a *state* of $\mathcal{P}$ is denoted by a tuple $(map, i, s, \sigma_1, \ldots, \sigma_m)$ where: (1) $map : [1, m] \rightarrow P$ is a mapping from threads $T_1, \ldots T_m$ to processes, (2) the currently active thread is T_i, $i \in [1, m]$, (3) s is a valuation of the shared variables, and (4) for each $j \in [1, m]$, σ_j is a *local state* of T_j. Observe that each such σ_j is composed of a valuation of the program counter, and of the local and global variables of the corresponding process, along with a *call-stack* of local variable valuations and program counters to model function calls.

At any state $(map, i, s, \sigma_1, \ldots, \sigma_m)$, the valuation of the shared variables s is referred to as the *shared state*. A *localized state* is the *view* of the state by the current process, i.e. it is $(\widehat{\sigma_i}, s)$, where $\widehat{\sigma_i}$ is the component of σ_i that defines the valuation of local and global variables, and the local pc (but not the call-stack), and s is the valuation of the shared variables in scope. Note that when a thread is not scheduled, its local state does not change.

The interleaved semantics of parameterized programs is given in the obvious way. We start with an arbitrary state, and execute the statements of `init` to prepare the initial shared state of the program, after which the threads become active. Given a state $(map, i, \nu, \sigma_1, \ldots, \sigma_m)$, it can either fire a transition of the process at thread T_i (i.e., of process $map(i)$), updating its local state and shared variables, or context-switch to a different active thread by changing i to a different thread-index, provided that in T_i we are not in a block of sequential statements to be executed atomically.

Reachability. Given a parameterized program $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^{n})$ and a target program counter *pc*, the *reachability problem* asks whether there exist an integer $m > 0$ and an execution of $\mathcal{P}$ that reaches a state $(map, i, \nu, \sigma_1, \ldots, \sigma_m)$ such that *pc* is the program counter of σ_i for some $i \in [1, m]$. Since two threads communicating through a finite shared memory suffice to simulate a Turing machine, this problem is clearly undecidable. Here we also consider the reachability under bounded-round schedules. For threads $T_1, \ldots, T_m$, a k-round schedule of $T_1, \ldots, T_m$ is a schedule that, for some ordering of such threads, activates them in k rounds, where in each round each thread is scheduled (for any number of events) according to this order. Observe that, restricting to executions under any

[1] Note: in this scheme, each thread creation causes a context-switch; true thread creation, without paying such cost (like in [1]), cannot be modeled in our framework.

k-round schedule does not place any bound on the number of threads which are involved. Given a $k \in \mathbb{N}$, the *reachability problem under bounded-round schedules* is the reachability problem restricted to consider only executions under k-round schedules.

3 Linear Interfaces

We now introduce the concept of linear interface, that captures the effect a block of threads has on the shared state, when involved in an execution of a k-round schedule. For the rest of the paper, we fix a parameterized program $\mathcal{P} = (S, \texttt{init}, \{P_i\}_{i=1}^{n})$ and a bound $k > 0$ on the number of rounds. We also use the notation $\overline{u}$ to refer to a tuple $(u_1, \ldots, u_k)$ of shared states of $\mathcal{P}$.

A pair of k-tuples of shared variables $(\overline{u}, \overline{v})$ is a *linear interface* of length k (see Figure 1) if: **(a)** there is an ordered block of threads $T_1, \ldots, T_m$ (running processes of $\mathcal{P}$), **(b)** there are k rounds of execution, where each execution starts from shared state u_i, exercises the threads in the block one by one, and ends with shared state v_i (for example, in Figure 1, the first round takes u_1 through s_1^1, t_1^1, s_2^1, t_2^1, ... to t_m^1 where the shared state is v_1), and **(c)** the local state of threads is preserved between consecutive rounds in these executions (in Figure 1, for example, t_1^1 and s_1^2 have the same local state). Informally, a linear interface is the *effect* a block of threads can have on the shared state in a k-round execution, in that they transform $\overline{u}$ to $\overline{v}$ across the block. Formally, we have the following definition (illustrated by Figure 1).

Definition 1. *(*Linear interface*) Let $\overline{u} = (u_1, \ldots, u_k)$ and $\overline{v} = (v_1, \ldots, v_k)$ be tuples of k shared states of a parameterized program $\mathcal{P}$ (with processes P). The pair $(\overline{u}, \overline{v})$ is a* linear interface *of $\mathcal{P}$ of length k if there is some number of threads $m \in \mathbb{N}$, an assignment of threads to processes $map : [1, m] \to P$ and states $s_i^j = (map, i, x_i^j, \sigma_1^{i,j}, \ldots, \sigma_m^{i,j})$ and $t_i^j = (map, i, y_i^j, \gamma_1^{i,j}, \ldots, \gamma_m^{i,j})$ of $\mathcal{P}$ for $i \in [1, m]$ and $j \in [1, k]$, such that, for each $i \in [1, m]$ and $j \in [1, k]$:*

- *$x_1^j = u_j$ and $y_m^j = v_j$;*
- *t_i^j is reachable from s_i^j using only local transitions of process $map(i)$;*
- *$\sigma_i^{i,1}$ is an initial local state for process $map(i)$;*
- *$\sigma_i^{i,j+1} = \gamma_i^{i,j}$ except when $j = k$ (local states are preserved across rounds);*
- *$x_{i+1}^j = y_i^j$, except when $i = k$ (shared states are preserved across context-switches of a single round);*
- *(t_i^j, s_{i+1}^j), except when $i = k$, is a context-switch.*

When $m = 1$, $(\overline{u}, \overline{v})$ is also called a thread linear interface. □

Note that the definition of a linear interface $(\overline{u}, \overline{v})$ places no restriction on the relation between v_j and u_{j+1}— all that we require is that the block of threads must take $\overline{u}$ as input and compute $\overline{v}$ in the k rounds, preserving the local configuration of threads between rounds.

Linear interfaces compose. Let $I = (\overline{u}, \overline{v})$ and $I' = (\overline{u}', \overline{v}')$ be two linear interfaces of length k. If the output of I matches the input of I', i.e., $\overline{v} = \overline{u}'$ holds, then the *composition* of I and I' is the pair $(\overline{u}, \overline{v}')$.

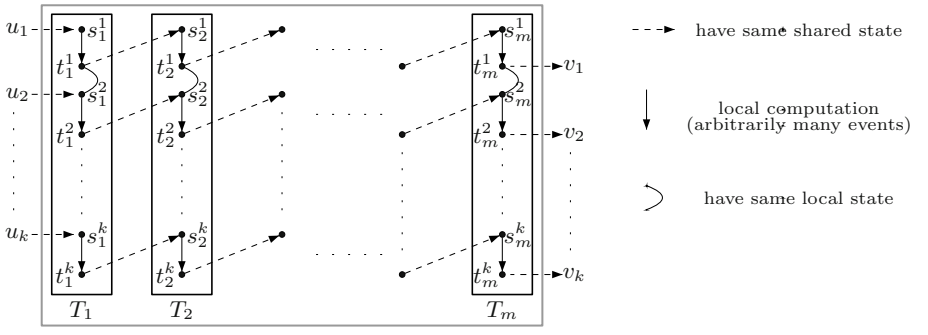


Fig. 1. A linear interface

Lemma 1. *The composition of linear interfaces of length k is a linear interface of length k. Moreover, each linear interface is either a thread linear interface or a composition of two or more thread linear interfaces.* □

An execution of a parameterized program under a k-round schedule can always be seen as a composition of thread linear interfaces that form a unique linear interface that have the following properties.

A linear interface $(\overline{u}, \overline{v})$ of length k is *wrapped* if $v_i = u_{i+1}$ for each $i \in [1, k-1]$. A linear interface $(\overline{u}, \overline{v})$ is *initial* if u_1, the first component of $\overline{u}$, is an initial shared state of $\mathcal{P}$. Thus, an execution of a parameterized program under a k-round schedule always corresponds to a wrapped initial linear interface $(\overline{u}, \overline{v})$. Such an execution is said to *conform* to $(\overline{u}, \overline{v})$. The following lemma is straightforward:

Lemma 2. *Let $\mathcal{P}$ be a parameterized program. An execution of $\mathcal{P}$ is under a k-round schedule iff it conforms to some wrapped initial linear interface of $\mathcal{P}$ of length k.* □

4 Reachability under Bounded-Round Schedules

In this section we give a fixed-point algorithm to solve the reachability problem under a bounded-round schedule for a parameterized program. From Lemma 2, it follows that all that is required is to compute, for a given parameterized program, all possible linear interfaces of size k, and then check among those that are both initial and wrapped. Since for a fixed k the number of linear interfaces of a program is finite, this can be computed as suggested by Lemma 1, starting with thread linear interfaces, and then composing them till a fixed-point is reached. However, it turns out that this does not work well in practice, as the computation of thread linear interfaces starts from arbitrary tuples of k shared states and then determines all the states reachable from them, and hence unreachable parts of the state-space can be explored. Early implementation results of this algorithm in fact failed miserably on our benchmarks. We now propose a more intricate algorithm that ensures that linear interfaces are computed and explored only on reachable states.

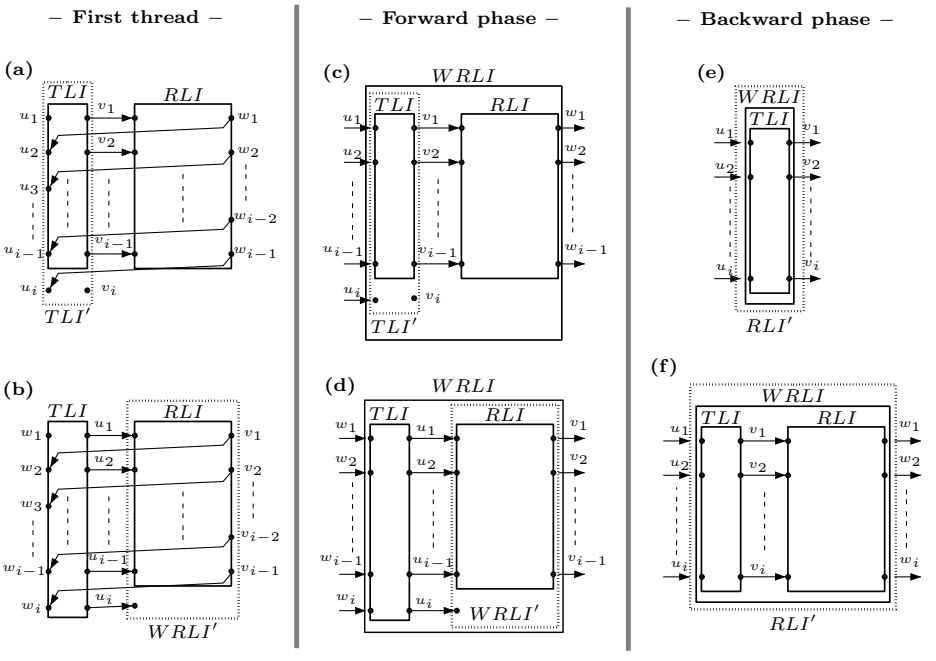


Fig. 2. Graphical representation of the update rules of the algorithm

Notation: Let π be an execution of $\mathcal{P}$ under a k-round schedule and $T_1, \ldots, T_m$ denote a block of threads scheduled consecutively in π. We say that π *covers* a linear interface $(\overline{u}, \overline{v})$ on $T_1, \ldots, T_m$ if along π, u_i matches the shared state on context-switching into T_1 and v_i matches the shared state on context-switching out of T_m in round i, for $i \in [1, k]$. Moreover, the localized state (σ, v_k) of T_m, which is visited along π when context-switching out of T_m in round k, is called a *final localized state* of $(\overline{u}, \overline{v})$ in π. A *right block* is a block of threads scheduled consecutively in the end of each round.

Description of the algorithm. The algorithm proceeds by computing for the input program, linear interfaces of size $1, 2, \ldots k$, ensuring that each is computed on reachable states only. In every iteration, we also compute the precise set of linear interfaces for right blocks (*right linear interfaces*).

Let us now describe, intuitively, how the i-th round is explored and how interfaces of length i are built when $i > 1$. We refer the reader to the diagrams in Fig. 2. In these diagrams, boxes drawn with solid lines denote interfaces that exist, while those with dotted lines denote new blocks that get created. Moreover, *TLI*, *RLI*, and *WRLI* refer to thread-linear interfaces, right linear interfaces, and "want blocks". Arrows denote equality of the shared states at the endpoints.

We start the i-th round with the first thread (see Fig. 2.a). We take an initial thread linear interface (*TLI*) of length $i-1$ and a right linear interface (*RLI*), still of length $i-1$, which composes with *TLI* and such that the resulting linear interface is both initial and wrapped. We then compute a localized state (σ, u_i),

where σ is from the final localized state of *TLI* and u_i is the shared state from the end of the $(i-1)$-th round in *RLI*. Using this, we can compute all possible states which are reachable by the thread in round i, and hence compute all the thread linear interfaces of length i covered by a run on the first thread (Fig. 2.a). Now the computation progresses on the second thread (see Fig. 2.b). For all the newly reached shared states u_i, we then create a *want block* (*WRLI*′) with the *RLI*'s input, the new input u_i, and the *RLI*'s output, which captures our desire that we want to continue the rest of the threads with this new input. Want blocks are not quite linear interfaces, as they have i inputs and $i-1$ outputs, but are crucial in guiding the computation.

Next, we enter the *forward phase* (Fig. 2.c and 2.d), where a want-block *WRLI*, a thread linear interface *TLI*, and a right linear interface *RLI* exist, and where the inputs of *WRLI* and *TLI* match, the outputs of *TLI* match the input of *RLI*, and the outputs of *WRLI* and *RLI* match. In this scenario, a new thread linear interface of size i is formed from *TLI* (inheriting the shared state from *WRLI* and the local state from *TLI*) and explored locally to form new thread linear interfaces of size i (Fig. 2.c). Further, these new thread linear interfaces create further want blocks to further the computation (Fig. 2.d).

Want blocks can also (non-deterministically) stop when the inputs precisely match the outputs, and create right linear interfaces (Fig. 2.e). This is the base case of the induction capturing the formation of right linear interfaces (starting from the last scheduled thread in each round) and starts the *backward phase*. This computation takes a right linear interface *RLI*, combines it with a thread linear interface *TLI* to the left of it, and provided a matching want block exists, combines them to form a larger right linear interface (Fig. 2.f). These computed right linear interfaces correspond to reachable blocks of computation (because we have checked them against want blocks, which were in turn reachable), and is used in the next iteration to ensure that only reachable states are explored.

Of course, the above three phases are not regulated sequentially, and are explored arbitrarily by fixed-point computations.

Fixed-point formulation. We formally describe our algorithm as a system of equations of the form $R = Exp$ where *Exp* is a positive boolean expression with first order quantification over relations and R is a relation which may also appear within *Exp* (*recursive* definition of relations is admitted).

In such equations, we will use the following base relations. *LocInit* and *ShInit* denote respectively the initial local states for each thread and the initial shared states (computed by executing the `init` block). $Wrap(\overline{u}, \overline{v})$ holds true if and only if $v_i = u_{i+1}$, for all $i \in [1, k-1]$. We also use $\langle$ *local reachability* $\rangle$ to denote a formula expressing the clauses of a fixed-point formulation of the states that are forward reachable using only transitions of a process. We omit the details on this formula since it is essentially the same as for sequential programs (see [13]).

Denote with $\mathcal{S}$ the following system of equations:

1. $TLI(i, \sigma, \overline{u}, \overline{v}) =$
$$(i = 1 \wedge LocInit(\sigma) \wedge u_1 = v_1 \wedge (ShInit(u_1) \vee \exists\, \overline{w}, \sigma'.\, TLI(1, \sigma', \overline{w}, \overline{u}))) \quad (1.1)$$
$$\vee\, (i > 1 \wedge u_i = v_i \wedge TLI(i-1, \sigma, \overline{u}, \overline{v}) \wedge \exists\, \overline{w}.\ (RLI(i-1, \overline{v}, \overline{w})$$

$$\wedge\ (\ (ShInit(u_1) \wedge Wrap(\overline{u},\overline{w})) \vee WRLI(i,\overline{u},\overline{w}))))) \quad (1.2)$$
$$\vee\ \langle\, local\ reachability \,\rangle \quad (1.3)$$

2. $WRLI(i,\overline{u},\overline{v}) = \quad i > 1 \wedge \exists \sigma, \overline{w}.$
$(TLI(i,\sigma,\overline{w},\overline{u}) \wedge RLI(i-1,\overline{u},\overline{v}) \wedge (\ WRLI(i,\overline{w},\overline{v}) \vee (ShInit(w_1) \wedge Wrap(\overline{w},\overline{v}))))$

3. $RLI(i,\overline{u},\overline{v}) = \quad (i = 1 \vee WRLI(i,\overline{u},\overline{v}))$
$\wedge \exists \sigma.(\ TLI(i,\sigma,\overline{u},\overline{v}) \vee (\exists \overline{w}.\ (TLI(i,\sigma,\overline{u},\overline{w}) \wedge RLI(i,\overline{w},\overline{v})))$

Observe that $\mathcal{S}$ is a system of positive equations. Thus by Tarski's fixed-point theorem, it has a unique least fixed-point, and the relations are well defined. The evaluation of $\mathcal{S}$ is graphically described in Fig. 2. After computing the above relations, the last step of our algorithm consists of evaluating the formula:

$$\varphi ::= \exists i, \sigma, \overline{u}, \overline{v}.(1 \leq i \leq k) \wedge TLI(i,\sigma,\overline{u},\overline{v}) \wedge Target(\sigma),$$

where the predicate $Target(\sigma)$ holds if and only if σ corresponds to a target program counter in the reachability query.

Correctness of the algorithm. The following lemma is crucial to prove our algorithm correct.

Lemma 3. *Let $\overline{u} = (u_1, \ldots, u_k)$, $\overline{v} = (v_1, \ldots, v_k)$, σ such that (σ, v_i) is a localized state of $\mathcal{P}$, $k \in \mathbb{N}$, and $i \leq k$.*

1. *$TLI(i,\sigma,\overline{u},\overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $(\overline{u}_i,\overline{v}_i)$ is a thread linear interface covered by π and (σ, v_i) is a final localized state of $(\overline{u}_i,\overline{v}_i)$.*
2. *$RLI(i,\overline{u},\overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $(\overline{u}_i,\overline{v}_i)$ is a right linear interface covered by π.*
3. *$WRLI(i,\overline{u},\overline{v})$ holds iff there is an execution π of $\mathcal{P}$ under a k-round schedule such that $T_1, \ldots, T_m$ are scheduled at the end of each round, u_i is the shared state on context-switching to T_1 along π in round i, $i > 1$, and $(\overline{u}_{i-1},\overline{v}_{i-1})$ is a right linear interface covered by π on $T_1, \ldots, T_m$.* □

Note that, when computing the fixed point of $\mathcal{S}$, the relations *TLI*, *RLI* and *WRLI* grow monotonically, and once a tuple is added, it is never removed from the set. Thus, from the lemma, we get that in our computation, we only explore the reachable state space of the parameterized program. Therefore, we have:

Theorem 1. *Given an integer $k \geq 0$, a parameterized program $\mathcal{P}$ and a program counter pc, pc is reachable in $\mathcal{P}$ under k-round schedules if and only if the formula φ is satisfiable. Moreover, while computing the least fixed-point of system $\mathcal{S}$, only reachable localized states of $\mathcal{P}$ are explored.* □

5 An Adequacy Check: Proving Program Correct

The algorithm to solve the reachability problem under a k-round scheduling, given in the previous section, can be used to show a parameterized program incorrect (when an error state is reached). However, when the algorithm's answer

is negative (i.e., an error state is not reachable) nothing can be inferred on the correctness of the input program. In this section, we present an *adequacy check* that attempts to make our verification scheme complete. In particular, for a parameterized program without recursive function calls, we design a test that gives a sufficient condition to show that the reachable states of the program under a k-round schedule are indeed all its reachable states. Though the proposed test is sound but incomplete, in next section, we show by reporting our experimental results that it is indeed quite effective in practice.

Fix a parameterized program $\mathcal{P}$ and $k \in \mathbb{N}$. We wish to ensure the following: "For any state s of $\mathcal{P}$, if s is reachable under a $(k+1)$-round schedule then it is also reachable under a k-round schedule" (*k-rounds-suffice condition*).

Note that checking this condition may be computationally hard, and hardness mostly resides in the fact that the number of threads in the executions under k-round schedules is a priori unbounded (and thus handling entire program states is by itself a problem). We propose a game-theoretic algorithm that refers to portions of states that are local to threads (localized states) and keeps summaries of the performed computation (linear interfaces), and thus avoids the need to refer to the entire state of the program, and parses it thread-by-thread.

In particular, we wish to define a two-player game G_k where player 0 (*Eve*) selects a state of $\mathcal{P}$ by revealing with each move a localized state which is visited along an execution under a $(k+1)$-round schedule in round $k+1$, and player 1 (*Adam*) attempts to match every move of Eve along an execution under the same schedule but in round k. A typical play in G_k is as follows.

Eve starts selecting a localized state λ_1 which is final for an initial thread linear interface I_1 of length $k+1$ (we recall that this means that there exists a program execution under a k-round schedule which covers I_1 and context-switch out of the first thread in round $k+1$ at λ_1). Then, Adam matches this move by showing that λ_1 is a final localized state of an initial thread linear interface L_1 of length k. The play continues with Eve selecting a final localized state λ_2 of a thread linear interface I_2 of length $k+1$ such that the output of I_1 matches the input of I_2. Then, Adam reacts by showing that λ_2 is also a final localized state of a thread linear interface L_2 of length k such that the output of L_1 matches the input of L_2. Let I_2' be the composition of I_1 and I_2, and L_2' be the composition of L_1 and L_2. In the next iteration, Eve makes a selection expanding over the next thread in the schedule the linear interface I_2', and similarly, Adam tries to matches this selection by expanding L_2', and so on until Adam cannot match a move of Eve. Then starting from this point till the end, only Eve is allowed to move and she will keep expanding the constructed linear interface as above.

A play is winning for Eve if she can select a sequence of moves that cannot be matched by Adam and doing so she can construct a wrapped and initial linear interface, thus proving that the selected localized states are indeed visited in the $(k+1)$-th round of an execution under a $(k+1)$-round schedule. Eve also wins if Adam matches all her moves, but the linear interface she constructs is wrapped while that by Adam is not. In all the other cases, Adam wins.

Technically, we can store in the states of the game the interfaces which are constructed by the two players and thus express such winning conditions as reachability goals. Also note, that fixing k, the size of G_k is bounded.

We can formally describe a decision algorithm to solve such a game using equations as in Section 4. In our formulation, we model a state of the game as a tuple of the form $s = (pl, in, al, \overline{u}, \overline{v}, \sigma, \overline{x}, \overline{y})$ where pl denotes the player which is in control of the state (0 for Eve and 1 for Adam), $in = 1$ iff player pl has not moved yet in the current play, $al = 1$ iff Adam is still in the play (i.e., he has matched all Eve's moves so far), $(\overline{u}, \overline{v})$ is the linear interface of length $k+1$ constructed by Eve in the play, (σ, v_{k+1}) is a final localized state of $(\overline{u}, \overline{v})$, and $(\overline{x}, \overline{y})$ is the linear interface of length k constructed by Adam.

The winning conditions can be captured with a predicate characterizing the winning states. To solve the game, the attractor-set based algorithm can be expressed using fixed points and therefore we can directly implement it in our formalism. Due to the lack of space, we only give here the details of the relation *E-move* which captures the moves of Eve (the relation for Adam being similar).

$$
\begin{aligned}
E\text{-}move(s, s') = \quad & (\ pl = 0 \wedge \overline{x}' = \overline{x} \wedge \overline{y}' = \overline{y} \wedge (\\
& (al = 1 \wedge pl' = 1 \wedge al' = 1 \wedge \\
& \quad ((in = 1 \wedge in' = 1 \wedge TLI(k+1, \sigma, \overline{u}', \overline{v}') \wedge ShInit(u_1)) && (1)\\
& \quad\ \vee (in = 0 \wedge in' = 0 \wedge \overline{u}' = \overline{u} \wedge TLI(k+1, \sigma', \overline{v}, \overline{v}')))) && (2)\\
& \vee (al = 0 \wedge pl' = 0 \wedge al' = 0 \wedge in = 0 \wedge \overline{u}' = \overline{u} \wedge TLI(k+1, \sigma', \overline{v}, \overline{v}')))) && (3)
\end{aligned}
$$

In the above formula, (1) corresponds to the first move of Eve in a play, (2) to her moves as long as Adam has matched all her previous moves, and (3) to her moves in the remaining cases (i.e., Adam has failed to match a move by Eve).

Observe that, if we restrict to parameterized programs where only non-recursive function calls are allowed, we can prove that if there is a winning strategy of Adam then the k-rounds-suffice condition holds, and therefore, there are no more reachable states to explore. However, the converse does not hold: if Eve has a winning strategy, we cannot conclude that considering executions under $(k+1)$-round schedules will allow us to discover new reachable states of the program. In fact, Eve could cheat by changing her selections depending on Adam's moves, and thus, even if a selected state is reachable within k rounds, Adam could fail to prove it. Thus, we have the following theorem:

Theorem 2. *Let $\mathcal{P}$ a parameterized program without recursive function calls. For all $k \in \mathbb{N}$, if the adequacy check holds then the k-rounds-suffice condition holds, and therefore all reachable states of $\mathcal{P}$ are visited in executions under k-round schedules.* □

6 Implementation and Experiments

Symbolic model-checker: We implemented a symbolic BDD-based model-checker for reachability in parameterized programs in a bounded number of rounds, as well as a symbolic *adequacy checker* that checks (soundly) whether k-round schedules reach all reachable states, using the tool framework GETAFIX [13]

Table 1. Experimental results

		2 thread Analysis		Parameterized Analysis 4 rounds			Parameterized Analysis unbounded rounds		
	#Bool. pgms.	Reach-able	Unreach-able	Reachable	Unreach-able	Time-out	Proved Unreach-able	Not proved unreachable (Pl.0 wins)	Time-out
i8xx_tco	765	460	305	314 (+13)	218	220	204	0	14
ib700wdt	492	330	162	208 (+13)	112	159	106	0	6
machzwd	568	341	227	274 (+23)	158	113	56	87	15
mixcomwd	429	276	153	213 (+23)	102	91	100	0	2
pcwd	256	171	85	171 (+0)	85	0	81	0	4
sbc60xxwdt	425	276	149	174 (+23)	94	134	92	0	2
sc1200wdt	491	299	192	200 (+13)	135	143	135	0	0
sc520_wdt	438	272	166	173 (+23)	104	138	15	89	0
smsc37b787_wdt	719	428	291	280 (+13)	140	286	140	0	0
w83877f_wdt	558	362	196	219 (+23)	103	213	15	88	0
w83977f_wdt	850	495	355	366 (+13)	126	345	125	0	1
wdt977	799	486	313	338 (+13)	127	321	125	0	2
wdt	533	348	185	221 (+17)	107	188	105	0	2
wdt_pci	892	800	92	378 (+23)	13	478	10	3	0
Total	**8215**	**5344**	**2871**	**3529 (+233)**	**1624**	**2829**	**1309**	**267**	**48**

that we have recently developed. Getafix allows writing BDD-based model-checkers using a high-level fixed-point calculus, without having to write low-level code. Getafix translates Boolean programs to logical formulas, implements heuristics for BDD orderings, and furnishes the model-checker designer with templates that capture the semantics of the program. High-level model-checking algorithms written in a fixed-point calculus get implemented by Getafix using the symbolic fixed-point model-checker called Mucke (see [13]).

We adapted Getafix to translate parameterized Boolean programs and handle DDVerify benchmarks. The algorithms for reachability in k rounds were implemented using the fixed-point formulas outlined in this paper. The adequacy check was also implemented using fixed-points: we captured the moves of player 0 and player 1 symbolically, and wrote a fixed-point backward attractor-based algorithm to solve the reachability game.

Experiments on device drivers: We subject our model-checker to a suite of Boolean programs derived from the `Lwatchdog` suite of drivers using the DDVerify tool [23], which abstracts Boolean programs from Linux device drivers, and also provides a fairly accurate Boolean model of the OS kernel. The model of the driver is obtained using predicate abstraction, and appropriate translations of Spinlocks, timer functions, and service routines that it may use. The kernel program models kernel code as well as other OS related behavior such as interrupts, etc. using non-determinism.

Each DDVerify benchmark consists of an OS kernel that interacts with a device driver. We obtained our concurrent models by taking *one* copy of the kernel module along with an *unbounded* number of copies of the device driver module. We subject our tool to about 8000 Boolean abstractions of 14 device drivers, abstracted to verify several (hundreds of) safety properties, at various levels of refinement.

The results are summarized in Table 1. The "2-thread analysis" columns report the number of Boolean programs that had an error-state reachable and those that did not, when considering just two threads, one modeling the OS and one modeling the driver (these results are identical to DDVerify).

We analyzed the programs using our parameterized analysis tool and searched the space reached within 4-round schedules for errors; the results are reported in the second set of columns. Note that even when an error state is reachable in the 2-thread analysis, it may not be reachable in the parameterized analysis (as the latter considers only a limited number of rounds); however this *never occurred* in our experiments. Similarly, note that when an error state is unreachable in the 2-thread analysis, it may be reachable in the parameterized analysis (as the latter considers an unbounded number of threads); this did happen in several examples, and is noted in parenthesis with a +-sign in the "Reach" column of the parameterized analysis (e.g., for the first set of drivers, the error state was reachable in 13 programs in the parameterized setting within 4 rounds, but not in the 2-thread setting). The parameterized analysis is computationally more expensive, and the model-checker ran out of resources (memory or time-out at 30sec) for the programs reported in the "Timeout" column.

The final set of columns report results for the *adequacy check* based on the reachability game on those programs that were unreachable in 4 rounds. The first column reports the number of programs our tool was able to prove entirely correct (any number of rounds and threads); the second column reports the number of programs that were not proved unreachable (this does not mean that the error state *is* reachable, as our adequacy check is not complete); and the last column gives the programs on which the tool ran out of resources (out of memory or reached time-out at 30sec). For example, in the first set of drivers, out of the 218 programs in which the error state was not reachable in 4 rounds, our tool was able to prove 204 of them completely correct, and 14 of them timed-out.

Observations from experiments: Several observations are in order:

- All error-states reachable in the 2-thread instantiation were found within 4 rounds in the parameterized system. This experimentally supports the conjecture that error-states are often reachable within a few rounds, even on Boolean program abstractions.
- There are several programs (~ 225) where a predicate abstraction that can prove a driver correct when working alone with the OS is not sufficient to prove it correct in the parameterized setting.
- Most interestingly, most programs (~ 1300 out of 1600, or $\sim 80\%$), when the error state was not reachable in 4 rounds, were proved entirely correct by our technique. In fact, our adequacy check was extremely effective in 11 of the 14 suites; 3 suites however have a significant percentage of programs that we were unable to prove entirely correct.

Note that a sound predicate abstraction followed by a successful parameterized verification proves the original driver correct for any number of threads and schedule; our tool achieves this for about 1300 instances.

References

1. Atig, M.F., Bouajjani, A., Qadeer, S.: Context-bounded analysis for concurrent programs with dynamic creation of threads. In: TACAS. LNCS, vol. 5505, pp. 107–123. Springer, Heidelberg (2009)
2. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
3. Clarke, E.M., Grumberg, O., Jha, S.: Veryfying parameterized networks using abstraction and regular languages. In: Lee, I., Smolka, S.A. (eds.) CONCUR 1995. LNCS, vol. 962, pp. 395–407. Springer, Heidelberg (1995)
4. Cobleigh, J.M., Giannakopoulou, D., Pasareanu, C.S.: Learning assumptions for compositional verification. In: Garavel, H., Hatcliff, J. (eds.) TACAS 2003. LNCS, vol. 2619, pp. 331–346. Springer, Heidelberg (2003)
5. Cohen, A., Namjoshi, K.S.: Local proofs for linear-time properties of concurrent programs. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 149–161. Springer, Heidelberg (2008)
6. Creese, S.J., Roscoe, A.W.: Data independent induction over structured networks. In: PDPTA.CSREA Press (2000)
7. Emerson, E.A., Kahlon, V.: Parameterized model checking of ring-based message passing systems. In: Marcinkowski, J., Tarlecki, A. (eds.) CSL 2004. LNCS, vol. 3210, pp. 325–339. Springer, Heidelberg (2004)
8. Emerson, E.A., Sistla, A.P.: Symmetry and model checking. In: Courcoubetis, C. (ed.) CAV 1993. LNCS, vol. 697, pp. 463–478. Springer, Heidelberg (1993)
9. Ghafari, N., Gurfinkel, A., Trefler, R.J.: Verification of parameterized systems with combinations of abstract domains. In: Lee, D., Lopes, A., Poetzsch-Heffter, A. (eds.) FMOODS 2009. LNCS, vol. 5522, pp. 57–72. Springer, Heidelberg (2009)
10. Ip, C.N., Dill, D.L.: Verifying systems with replicated components in murphi. Formal Methods in System Design 14(3), 273–310 (1999)
11. Kesten, Y., Maler, O., Marcus, M., Pnueli, A., Shahar, E.: Symbolic model checking with rich assertional languages. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 424–435. Springer, Heidelberg (1997)
12. Kesten, Y., Pnueli, A., Shahar, E., Zuck, L.D.: Network invariants in action. In: Brim, L., Jančar, P., Křetínský, M., Kucera, A. (eds.) CONCUR 2002. LNCS, vol. 2421, pp. 101–115. Springer, Heidelberg (2002)
13. La Torre, S., Madhusudan, P., Parlato, G.: Analyzing recursive programs using a fixed-point calculus. In: PLDI, pp. 211–222. ACM, New York (2009)
14. La Torre, S., Madhusudan, P., Parlato, G.: Reducing context-bounded concurrent reachability to sequential reachability. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 477–492. Springer, Heidelberg (2009)
15. La Torre, S., Madhusudan, P., Parlato, G.: Model-checking Parameterized Concurrent Programs using Linear Interfaces IDEALS Technical Report, University of Illinois, http://hdl.handle.net/2142/15410
16. Lahiri, S.K., Bryant, R.E.: Predicate abstraction with indexed predicates. ACM Trans. Comput. Log. 9(1) (2007)
17. Lal, A., Reps, T.W.: Reducing concurrent analysis under a context bound to sequential analysis. In: Gupta, A., Malik, S. (eds.) CAV 2008. LNCS, vol. 5123, pp. 37–51. Springer, Heidelberg (2008)
18. Musuvathi, M., Qadeer, S.: Iterative context bounding for systematic testing of multithreaded programs. In: PLDI, pp. 446–455. ACM, New York (2007)

19. Pnueli, A., Xu, J., Zuck, L.D.: Liveness with (0, 1, ∞)-counter abstraction. In: Brinksma, E., Larsen, K.G. (eds.) CAV 2002. LNCS, vol. 2404, pp. 107–122. Springer, Heidelberg (2002)
20. 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)
21. Qadeer, S., Wu, D.: Kiss: keep it simple and sequential. In: PLDI, pp. 14–24. ACM, New York (2004)
22. Suwimonteerabuth, D., Esparza, J., Schwoon, S.: Symbolic context-bounded analysis of multithreaded java programs. In: Havelund, K., Majumdar, R., Palsberg, J. (eds.) SPIN 2008. LNCS, vol. 5156, pp. 270–287. Springer, Heidelberg (2008)
23. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE, pp. 501–504. ACM, New York (2007)

Dynamic Cutoff Detection in Parameterized Concurrent Programs*

Alexander Kaiser, Daniel Kroening, and Thomas Wahl

Oxford University Computing Laboratory, United Kingdom

Abstract. We consider the class of finite-state programs executed by an unbounded number of replicated threads communicating via shared variables. The *thread-state reachability* problem for this class is essential in software verification using predicate abstraction. While this problem is decidable via *Petri net coverability* analysis, techniques solely based on coverability suffer from the problem's exponential-space complexity. In this paper, we present an alternative method based on a *thread-state cutoff*: a number n of threads that suffice to generate all reachable thread states. We give a condition, verifiable dynamically during reachability analysis for increasing n, that is sufficient to conclude that n is a cutoff. We then make the method complete, via a coverability query that is of low cost in practice. We demonstrate the efficiency of the approach on Petri net encodings of communication protocols, as well as on non-recursive Boolean programs run by arbitrarily many parallel threads.

1 Introduction

Concurrent software is gaining tremendous importance due to the shift towards multi-core computing architectures. The software is executed by parallel threads, in an asynchronous interleaving fashion. The most prominent and flexible model of communication between the threads is the use of fully shared variables. This model is supported by well-known programming APIs, e.g. the POSIX pthread model and Windows' WIN32 API. Bugs in programs written for such environments tend to be subtle and hard to detect by means of testing, strongly motivating formal analysis techniques.

In this paper, we consider the case in which no a-priori bound on the number n of concurrent threads is known. This scenario is most relevant in practice; it applies, for example, to a server that spawns additional worker threads in response to a high work load. We focus here on *replicated finite-state* programs: the program itself only allows finitely many configurations, but is executed by an unknown number of threads, thus generating an unbounded state space. An important practical instance of this scenario is given by non-recursive concurrent Boolean programs. Boolean program verification is a bottleneck in the widely-used predicate abstraction-refinement framework.

We tackle in this paper the *thread-state reachability* problem. A thread state is defined as a valuation of the shared program variables, plus the local state

* This research is supported by the EPSRC project EP/G026254/1.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 645–659, 2010.

of one thread. Thread-state reachability is routinely used to encode multi-index safety properties of systems, such as mutually exclusive access to some resource.

The thread-state reachability problem for replicated finite-state programs is equivalent in complexity to the *coverability problem* for Petri nets. The latter problem is decidable [18], but has an exponential lower space bound [7]. In order to not fall victim to this complexity, the approach presented in this paper takes advantage of widely accepted empirical evidence that often a small number of threads suffice to exhibit all relevant behavior that may lead to a bug. If this number is efficiently computable, the unbounded thread-state reachability problem reduces to a finite-state exploration problem, for which quite efficient engines have recently emerged [3].

To be more precise, for every finite-state program $\mathbb{P}$, there is a number c such that any thread state reachable for *some* number of threads running $\mathbb{P}$ can in fact be reached given c threads. We call such a number a *thread-state cutoff* of $\mathbb{P}$. Previous results on computing cutoffs of a program $\mathbb{P}$ tend to either restrict the communication scheme [12,17], or yield cutoffs that are polynomial in the number of states of $\mathbb{P}$ [11]. Both types are inapplicable to Boolean program verification, since concurrent programming APIs rely on very liberal shared-variable communication, while a Boolean program $\mathbb{P}$ typically has millions of states, rendering even linear-size cutoffs useless.

In contrast to previous solutions, we give in this paper a condition on a number n whose satisfaction allows us to conclude that n is the cutoff of a program $\mathbb{P}$. To obtain such a condition, we first show that, if n is *not* the cutoff, then there exists a number $n' > n$ and a thread state reachable in the n'-thread system $\mathbb{P}_{n'}$ whose reachability requires a particular conducive constellation of several threads in $\mathbb{P}_n$. If the reachable states in $\mathbb{P}_n$ do not permit such a constellation, then n is indeed the cutoff of $\mathbb{P}$.

We then turn this idea into a complete and *tight* cutoff detection algorithm. Completeness is achieved using backward coverability analysis to rule out the reachability of the thread states identified as candidates for the constellation mentioned above. We argue that these candidate state are *benign*, in that backward coverability analysis on them is efficient and does not defeat the original purpose of avoiding such analysis. Minimality of the cutoff is ensured by applying the cutoff detection method iteratively to the values $n = 1, 2, \ldots$. Since our method uses reachability information, we speak of *dynamic* cutoff detection.

We experimentally investigate the cutoffs of a large number of Petri net and Boolean program examples, modeling concurrent systems of various types. We demonstrate the superiority of our cutoff method over several earlier algorithms based solely on Petri net coverability. Our experiments showcase the method as a very promising algorithmic solution to coverability problems for Petri nets, and as an efficient technique for thread-state reachability analysis in realistic, if non-recursive, Boolean programs run by arbitrarily many threads.

2 Basic Definitions

Let $\mathbb{P}$ be a program that permits only finitely many configurations. In particular, $\mathbb{P}$'s variables are of finite range, and the function call graph, if any, of $\mathbb{P}$ is acyclic. An instance of the class of programs $\mathbb{P}$ is given by non-recursive Boolean programs, which are obtained from `C` programs using predicate abstraction. The use of Boolean programs as abstractions of `C` programs was promoted by the success of the SLAM project [1]. We use concurrent Boolean programs in the experimental evaluation of our approach and refer the reader to [8] for a detailed description.

Program $\mathbb{P}$ gives rise to a family $(M_n)_{n=1}^{\infty}$ of *replicated finite-state system* models as follows. $\mathbb{P}$'s variables are declared to be either *shared* or *local*. A valuation of the shared variables is called a *shared state*, a valuation of the local variables is called a *local state*. M_n is a Kripke structure modeling an n-thread concurrent program. The states of M_n have the form $(s, l_1, \ldots, l_n)$, where s is a shared state and l_i is a local state; we say l_i is the local state *of thread* i. M_n's execution model is that of interleaved asynchrony. That is, the set of transitions of M_n is the set of pairs of the form

$$\left((s, l_1, \ldots, l_{i-1}, l_i, l_{i+1}, \ldots, l_n) \quad , \quad (s', l_1, \ldots, l_{i-1}, l_i', l_{i+1}, \ldots, l_n) \right) \qquad (1)$$

such that $(s, l_i) \rightarrow (s', l_i')$ corresponds to a statement in $\mathbb{P}$. Only one thread, i, can make a step at a time. A step may change the local state of that thread and the shared state; we call thread i *active* in the transition. The pair (s, l_i) is called the *thread state* of thread i in global state $(s, l_1, \ldots, l_n)$; a thread state summarizes the part of the global state that is accessible to a thread. A thread has neither read nor write access to local variables of other threads. Note that if a transition changes the shared state of M_n (i.e., $s \neq s'$), it changes the thread state of *every* thread of M_n. Such transitions capture thread communication.

In order to define the thread-state reachability problem considered in this paper, let T be the (finite) universe of thread states, i.e., pairs of shared and local variable valuations, **irrespective of** n. A state $(h, l_1, \ldots, l_n)$ of M_n *contains* thread state (s, l) if $h = s$ and, for some i, $l_i = l$. Thread state t is *reachable in* M_n if there exists a reachable global state of M_n that contains t; reachability of global states in M_n is defined with respect to some set of initial states as usual. We denote the set of thread states reachable in M_n by R_n, and the set $\cup_{n=1}^{\infty} R_n$ of thread states reachable for *some* number of threads by R. Note that, for any n, $R_n \subseteq R \subseteq T$; in particular, these reachability sets are finite. The *thread-state reachability problem* is now defined as follows: given $\mathbb{P}$, determine R.

Our model of replicated finite-state system families $(M_n)_{n=1}^{\infty}$ formalizes classical *parameterized* systems, where the number of running threads is fixed upfront but unknown. Our techniques apply equally to systems where the number of threads can change at runtime. It is quite easy to show that the two models are equivalent for reachability properties. Further, our techniques extend to the case of multiple program templates, as in a heterogeneous synchronization problem with arbitrarily many *readers* and *writers*. For simplicity, we focus in the rest of this paper on the single-template parameterized case formalized above.

3 Background: Decidability of Thread-State Reachability

The thread-state reachability problem as defined in the previous section is decidable, via a reduction to the *coverability problem* for *vector addition systems with states* (VASS), as follows. A VASS is a finite-state machine whose edges are labelled with integer vectors of some fixed dimension. A *configuration* of a VASS is a pair (q, x) where q is a state and x is a vector of **non-negative** integers. There is a transition $(q, x) \to (q', x')$ if there is an edge $q \xrightarrow{v} q'$ in the VASS such that $x' = x + v$; symbol $+$ denotes pointwise addition. Given an initial configuration (q_0, x_0), a configuration (q, x) is *reachable* if there exists a sequence of transitions starting at (q_0, x_0) and ending at (q, x). The *coverability problem* asks whether a given configuration (q, x) is *covered* by the reachable configurations of the VASS, i.e., whether a configuration (q, x') is reachable such that $x' \geq x$, where $\geq$ is defined pointwise.

Theorem 1 ([18]). *The coverability problem for VASS is decidable.*

The decision procedure by Karp and Miller [18] builds a rooted tree that represents the set of covered configurations of a vector addition system. Unfortunately, it operates not even in primitive-recursive space. In response to this daunting complexity, alternative algorithms exploring *well-structured transition systems* (WSTS), of which VASS are an example, have been developed [13,15]. Their efficiency is handicapped by the EXPSPACE lower bound of the coverability problem, the proof of which is attributed to Cardoza, Lipton and Meyer [7].

Replicated finite-state systems as vector addition systems. Using the components of a vector to count the number of threads in each of the possible local states, a VASS can simulate a replicated finite-state system: a thread transition $(s, l) \to (s', l')$ is represented by a VASS edge $s \xrightarrow{v} s'$ such that the l-th component of v is -1, the l'-th component is 1, and all others are 0. A thread state (s, l) of the program is reachable in the program's concurrent execution exactly if there is a reachable VASS configuration (s, x) such that the l-th component of x is at least 1. By definition, this is the case exactly if the VASS configuration (s, x_0) is *covered*, where x_0 is all-zero except the entry at position l, which is 1. The latter problem is decidable by Theorem 1. We obtain:

Corollary 2. *The thread-state reachability problem for replicated finite-state programs is decidable.*

It can be shown that the VASS coverability problem is, conversely, reducible to the thread-state reachability problem, in a way that makes the thread state reachability problem EXPSPACE complete as well. We remark that all reduction results sketched in this section hold equivalently for Petri nets in place of vector addition systems. Since the former are of a somewhat greater practical appeal, we will use Petri nets and their tools as a reference point in the experimental Section 6 later in this paper.

4 Thread-State Reachability via Cutoffs

Our computational model, according to which the possible transitions of a thread are determined only by its local state and the shared state, guarantees the following monotonicity property:

Property 3. *Sequence* $(R_n)_{n=1}^{\infty}$ *is monotone in* n*:* $n_1 \leq n_2$ *implies* $R_{n_1} \subseteq R_{n_2}$.

This property holds since every path in M_{n_1} can be extended to a path in M_{n_2} of the same length by adding $n_2 - n_1$ thread components to each state along the path and letting the new threads idle in their initial state.

Sequence $(R_n)_{n=1}^{\infty}$ thus never decreases. Since, on the other hand, the set R of reachable thread states is finite and $R_n \subseteq R$ for every n, the sequence can increase only a finite number of times. This implies that, for every finite-state program $\mathbb{P}$, there is a number c such that any reachable thread state can in fact be reached given c threads. Such a number is called a (thread-state) cutoff.

Definition 4. *A* ***thread-state cutoff*** *(or* ***cutoff*** *for short) for family* $(M_n)_{n=1}^{\infty}$ *is a number* $c \in \mathbb{N}$ *such that, for all* $n \geq c$*,* $R_n = R_c$*.*

In particular, we have $R_c = R$. Knowing the cutoff would therefore allow us to compute the set of reachable thread states using an efficient *finite-state model checker*. In order to turn this possibility into a viable alternative to coverability methods, we not only have to find means of computing the cutoff efficiently. We also need the minimum cutoff c_0 to be small enough that a model checker can compute R_{c_0} with reasonable resources.

The minimum cutoff of a finite-state program can in principle be arbitrarily large: given a number c, consider the following program with a shared variable $s \in \{0, \ldots, c\}$, initially 0.

```
0: s := s + 1 (mod c+1)
1: if s = c: error
```

This program has a minimum cutoff of c. There is, however, widely accepted (although, to our knowledge, rarely documented) empirical evidence that, in "typical" parameterized programs, a small number of threads suffice to exhibit all relevant behavior that may lead to a bug. We will be able to gauge the precision of this claim in the experimental Section 6 at the end of the paper. For now, we return to our main objective: determining cutoffs efficiently in practice.

5 Determining Thread-State Cutoffs

Emerson and Kahlon present several results for *statically* obtained cutoffs that are linear in the size of the program template (such as a Kripke model of a Boolean program) [11]. While valuable in establishing the decidability of certain fragments of the parameterized model checking problem, such cutoffs are unlikely to be of practical value in our context, since they are often not *tight* and in fact

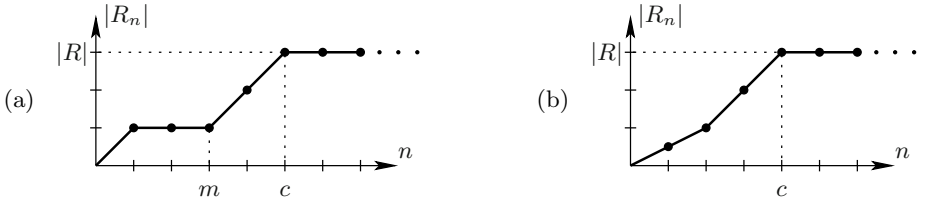


Fig. 1. (a) An intermediate plateau; (b) a strictly monotone thread state sequence

vastly overapproximate the minimum number of threads needed to reach all reachable thread states.

We propose in this paper a *dynamic* method to determine the cutoff. That is, instead of pre-computing the cutoff for the family, we *detect* it during the reachability analysis on the systems M_n, for increasing values of n. Our first contribution will be a condition that, based on certain observations on the reachability result obtained for M_n, allows us to conclude that we do not need to increase n further. Such a method has the potential of finding cutoffs that are orders of magnitude smaller than those computed by the static techniques.

5.1 Thread-State Sequences with Plateaus

Consider the thread-state sequence $(R_n)_{n=1}^{\infty}$ and a value m at which the sequence *plateaus*, i.e. $R_m = R_{m-1}$. It is tempting to conclude that a cutoff has been found when this happens. This temptation is fallacious, however, as the sequence of reached thread states may resume growth for thread counts exceeding m, even after several steps of plateauing.

Definition 5. *Value m is a **plateau endpoint** of (R_n) if $R_{m-1} = R_m \subsetneqq R_{m+1}$.*

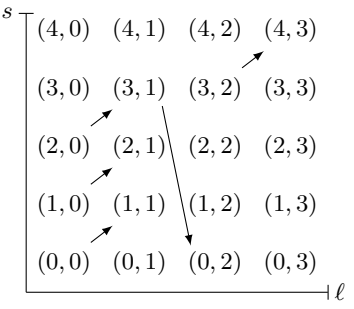
This situation is depicted in Figure 1 (a). The fallacious argument mentioned above would only be valid if every thread-state sequence was *strictly monotone* up to the minimum cutoff c, as shown in Figure 1 (b). A system with an intermediate plateau is induced by the finite-state program given in Figure 2. It can be synthesized into a four-line Boolean program with three shared variables.

Let us investigate the somewhat unintuitive phenomenon of intermediate plateaus more closely. Recall that if a transition changes the shared state of the program, the thread state of *every* thread is affected. As a result, a thread that is not itself active in the transition may reach a new thread state. We say that such a thread state is reached *passively*.

This situation is shown in Figure 3 (a). Thread i is active and changes, in addition to its local state, the shared state from r to s (solid line). As a side effect, thread state (s, h_j) is reached passively (dashed line). Note that the local state of thread j remains at h_j. Figure 3 (b) is a special case of (a) where threads i and j happen to reside in the same local state $h_i = h_j$ before i executes.

Returning to the issue of intermediate plateaus: one can show that, if m is a plateau endpoint, there exists a thread state in $R_{m+1} \setminus R_m$ that is reached passively. We will see next that in fact a much stronger statement holds.

(a)



(b)

n	R_n
1	$\{(0,0),(1,1)\}$
2	$R_1 \cup \{(1,0),(2,1)\}$
3	$R_2 \cup \{(0,1),(0,2),(2,0),(3,1)\}$
4	$R_3 \cup \{(1,2),(3,0)\}$
5	$R_4 \cup \{(2,2)\}$
6	$R_5 \cup \{(3,2),(4,1),(4,3)\}$
7	$R_6 \cup \{(4,0)\}$
8	$R_7 \rightarrow$ **plateau endpoint**
9	$R_8 \cup \{(4,2)\} \rightarrow$ **cutoff**

Fig. 2. (a) A finite-state program over variables (s, ℓ) with initial state $(0,0)$; (b) the thread-state sequence induced by (a), exhibiting a plateau of length 1

5.2 A Sufficient Cutoff Condition

Equipped with the considerations from Section 5.1, we can now derive a sufficient cutoff condition for thread-state reachability. Technically, we will establish instead a *necessary* condition for m *not being* a cutoff. The following lemma is the crucial insight.

Lemma 6. *Suppose m is **not** a cutoff for family $(M_n)_{n=1}^{\infty}$. Let $m' = \min\{n : R_n \supsetneq R_m\}$, and let t be a thread state in $R_{m'} \setminus R_m$ with minimum distance from the initial state set. Then t is reached **passively**.*

Proof. Let i be the thread active during the global transition of $M_{m'}$ when t is first reached. We have to show that t is *not* reached by thread i.

To this end, let $t_1 \rightarrow t_2$ be the thread transition executed by thread i that causes t to be reached by some thread; we prove $t \neq t_2$. Transition $t_1 \rightarrow t_2$ happens in $M_{m'}$, so $t_1 \in R_{m'}$. Since t_1 has shorter distance to the initial state set than t_2 and thus than t, we conclude $t_1 \notin R_{m'} \setminus R_m$, thus $t_1 \in R_m$. This in turn implies $t_2 \in R_m$, since the set R_m is closed under thread transitions: any path in M_m to a state containing t_1 can be extended, via $t_1 \rightarrow t_2$, to a path in M_m to a state containing t_2. Since $t \notin R_m$, it follows $t_2 \neq t$. □

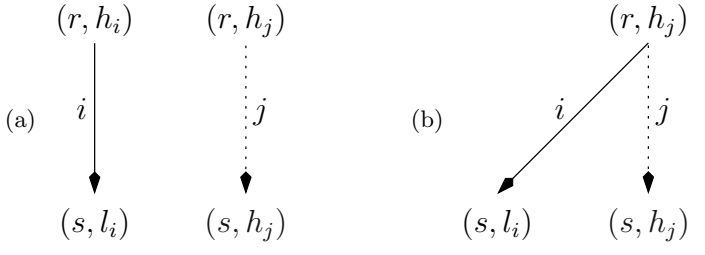


Fig. 3. (a) The general and (b) a special case of reaching thread state (s, h_j) passively

We can exploit this lemma as follows to derive a necessary is-not-the-cutoff condition. If m is not the cutoff, then the first new thread state encountered for $n > m$ is in fact reached passively. The ability to reach a thread state passively requires a constellation of reachable thread states as shown in Figure 3 (a), where the new thread state is denoted (s, h_j). We now observe that the thread states (r, h_i), (r, h_j) and (s, l_i) mentioned in the figure are all members of the current reachability set R_m. To see this, note that (r, h_i) and (r, h_j) are reached *before* (s, h_j). Since (s, h_j) has minimum distance, among all new thread states, we conclude that (r, h_i) and (r, h_j) are not new and are thus elements of R_m. Thread state (s, l_i) is an element of R_m since it is a direct successor of (r, h_i). We summarize: if m is not the cutoff, there exist three thread states (r, h_i), (r, h_j) and (s, l_i) in R_m such that

- $(r, h_i) \to (s, l_i)$ is a valid thread transition according to $\mathbb{P}$, and (2)
- $(s, h_j) \notin R_m$. (3)

We call thread states (r, h_i), (r, h_j) and (s, l_i) in R_m with these properties a *candidate triple*. If no candidate triple can be found, no thread state can possibly be reached passively in the future. Together with Lemma 6, we obtain:

Corollary 7. *Suppose no candidate triple exists in R_m. Then m is a cutoff for family $(M_n)_{n=1}^{\infty}$.*

We refer to the check of absence of candidates as the *cutoff test*. Unlike Lemma 6, the test conditions depend only on the program $\mathbb{P}$ and on M_m. The downside is that the cutoff test is incomplete for cutoff detection. To see this, consider the finite-state program over the state space $\{0, 1, 2\} \times \{0, 1\}$ with initial state $(0, 0)$ and the two transitions

$$(0, 0) \to (1, 1) \qquad \text{and} \qquad (1, 1) \to (2, 0)\,.$$

This program induces a parameterized family $(M_n)_{n=1}^{\infty}$ where the cutoff test fails for every n: the candidate triple $(1, 1)$, $(1, 1)$, $(2, 0)$ never vanishes. (The triple happens to be of the special form of Figure 3 (b).) We will fix this problem in the following section.

5.3 Sound, Complete and Tight Cutoff Detection

The cutoff test ignores that, in order to give rise to the new thread state (s, h_j), the candidate triple must be *realizable*: there must exist an n and a global state reachable in M_n that contains **both** (r, h_i) and (r, h_j). In the example in Section 5.2, no two threads can simultaneously enter a state of the form $(-, 1)$. It turns out that realizability of candidate triples precisely characterizes cutoffs:

Theorem 8. *Thread count m is a cutoff for family $(M_n)_{n=1}^{\infty}$* ***exactly if*** *R_m contains no realizable candidate triples.*

Proof. (i) If m is a cutoff and the candidate thread states (r, h_i) and (r, h_j) are, for some $n \geq m$, simultaneously reachable, then thread state (s, l_j) becomes reachable when M_n is analyzed. Since $(s, l_j) \notin R_m$ by equation (3), (s, l_j) is new, contradicting the stipulation that m is a cutoff.

(ii) If m is not a cutoff, then, by Lemma 6, there exists a passive thread transition that reaches a thread state unreachable in M_m. As shown in the proof of that lemma, the three thread states participating in the reaching of the new thread state all belong to R_m and thus form a candidate triple. For the passive transition to actually happen (the lemma proves that it does), the thread states (r, h_i) and (r, h_j) must be simultaneously reachable. So there exists at least one realizable candidate triple. □

Simultaneous reachability of (r, h_i) and (r, h_j) in the family $(M_n)_{n=1}^{\infty}$ cannot be checked by looking only at R_m. We will use *backward coverability* analysis for this step. The candidates represent a *minimal* set of thread states whose unrealizability guarantees the cutoff property. This minimality gives rise to the hope that candidates can be reachability-checked more efficiently than arbitrary thread states. We measure the cost of this check in detail in Section 6, using the MIST tool set [13] as the coverability engine.

Putting the cutoff test and this analysis together, we obtain Algorithm 1 for cutoff detection. The algorithm maintains the invariant that, at entry to the loop in Line 2, the reachability set R_n is guaranteed to have been computed, for the current value of n. In Line 2, the algorithm starts two computational threads in parallel. The first, **A**, computes the candidate triples for R_n. If any of them is realizable, which is checked using backward coverability analysis, we know by Theorem 8 that n is not a cutoff. The thread aborts, and control proceeds to Line 3. If no triple is realizable (or there are no candidates), we return that n is the cutoff; this terminates the algorithm.

The second thread, **B**, computes the next reachability set R_{n+1}, using a finite-state forward search. This is done in parallel with the candidate check since, as soon as we know that $R_{n+1} \supsetneq R_n$, we can abort the candidate check in thread **A**: we know that n is not the cutoff. If $R_{n+1} = R_n$, thread **B** terminates normally.

In Line 3, the main thread synchronizes the computation by waiting for the termination (or abortion) of **A** and **B**. This is crucial since the set R_{n+1} needs to be available in the next round. We then increase n and re-enter the loop. Note that if the backward analysis in round n reveals that some triple $\mathcal{T}$ is realizable, we do not know for which value $n' > n$ this will happen. As a result, intermediate plateaus of the sequence $(R_n)_{n=1}^{\infty}$ cannot be short-circuited.

Correctness. Termination of the algorithm follows immediately from Theorem 8: Suppose n is the cutoff. Then any candidate triples in R_n are not realizable, so thread **A** returns "cutoff n". Note that $R_{n+1} = R_n$, so thread **B** does not abort **A**. Theorem 8 similarly guarantees partial correctness. The combination of termination and partial correctness guarantees that Algorithm 1 returns in fact the *minimum* cutoff c_0: it does not terminate for $n < c_0$, by the partial correctness. It never reaches $n > c_0$, since it terminates for $n = c_0$.

Algorithm 1. Cutoff detection

Input: system family $(M_n)_{n=1}^{\infty}$

1: $n := 1$; compute R_1 // finite-state

2: **A**:
```
compute set C_n of cand. triples
if ∃T ∈ C_n: T realizable
   abort A
return "cutoff n"
```
||
B:
```
compute R_{n+1} // finite-state
if R_{n+1} ⊋ R_n
   abort A
```

3: sync(**A**,**B**)

4: $n := n + 1$; **goto** 2

Implementation. We illustrate how to compute candidate triples (first step of thread **A** in Algorithm 1). First note that conditions (2) and (3) on the candidates (r, h_i), (r, h_j) and (s, l_i) imply all of the following:

- $r \neq s$ (since $(r, h_j) \in R_m$, but $(s, h_j) \notin R_m$)
- $l_i \neq h_j$ (since $(s, l_i) \in R_m$, but $(s, h_j) \notin R_m$)
- (r, h_i), (r, h_j) are not simultaneously reachable in M_m (since otherwise $(s, h_j) \in R_m$, passively)

To compute the candidates, we iterate over pairs of thread states (r, h_i), (r, h_j) in R_m that are not simultaneously reachable in M_m (this information is taken from the reachable global states set of M_m), and select successor thread state (s, l_i) by consulting the program text under the additional constraints that $r \neq s$ and $l_i \neq h_j$. The remaining condition $(s, h_j) \notin R_m$ can be tested efficiently, say by storing R_m in a sorted container or a hash table.

6 Experimental Evaluation

We implemented two variants of Algorithm 1. The first is our Petri net coverability checker, eCUT, which we tested on 23 Petri nets examples from diverse programming domains. The second is our symbolic thread-state reachability checker for Boolean programs, sCUT, which we tested on 852 Boolean programs, generated from Linux device driver code. The Petri nets induce relatively small state spaces, but exhibit challenging concurrent behavior. In contrast, the Boolean programs induce huge state spaces, but exhibit rather simple concurrency. All experiments were performed on a 16GB/3GHz Intel Xeon machine running the 64-bit variant of Linux 2.6 with a 45min timeout.

6.1 Petri Net Coverability

Our coverability checker eCUT forward-computes an explicit-state representation of the sets R_n, and uses the backward search engine of the Mist toolset to check candidates for reachability. We evaluate eCUT using 5 bounded and 18 unbounded Petri nets, ranging from concurrent production systems and communication protocols to broadcast protocols. Each net is transformed into a

Table 1. Results of ECUT on Petri net benchmarks. S, L, T: # shared states, local states, thread transitions; $\sum fw$, $\sum bw$, ECUT: time for forward searches, backward searches, total ECUT runtime in seconds; c: cutoff (if unsafe: #threads until error); $|R_c|/|C_c|$: # reachable thread states/# candidate triples at bound c.

Benchmark	S	L	T	$\sum fw$	$\sum bw$	ECUT	c	$\lvert R_c\rvert$	$\lvert C_c\rvert$	Result
Readwrite	24	14	33	0.01	0.2	0.2	9	198	29	safe
Mesh2x2	35	33	71	0.6	0.01	0.8	9	844	40	safe
Multip.	20	19	45	0.1	0.8	0.9	8	257	10	safe
Pncsa	37	32	73	1.4	0.1	1.5	7	860	122	unsafe
Fms	26	23	49	0.6	1.2	1.6	12	361	5	safe
Bh250	507	254	1,009	0.6	6.0	6.7	3	1,768	31,875	safe
Mesh3x2	55	53	115	277.4	1.2	278.6	13	2,228	67	safe
Kanban	29	17	49	–	–	–	–	–	–	mem-out

replicated finite-state system. Transitions are split into sequences of thread transitions using fresh intermediate shared states. Given p places and t transitions, this required $p + 1$ local states, $1.2t$ shared states and $2.2t$ thread transitions on average. The original coverability property translates into the reachability of a suitable thread state. All examples and correctness properties are from [13] and [4].

Within 5min or much less, ECUT succeeds on 22 examples (21 safe, 1 unsafe), and memory-outs on 1. Table 1 shows details of the analysis; we omit instances with runtimes below 0.2s and only show the most challenging from [4], namely Bh250. The *Kanban* example has a cutoff beyond 20; our implementation reaches the memory limit after 10min and more than $6 \cdot 10^7$ explored states in round $n = 15$. In these examples, neither the finite-state forward nor the backward search dominate the overall runtime, advocating the use of a combination of both.

Comparison with other algorithms. We compare our implementation with four algorithms implemented in the Petri net coverability tool set Mist [13]: a pure backward search (BW) (the same we use to check candidates), and three abstraction refinement schemes. The latter combine infinite-state forward and backward search, using abstractions that minimize the number of predicates used to encode places (TSI, EEC) or the dimensionality of the Petri net (IC4PN).

Figure 4 shows the number of instances the different algorithms can solve within 45min/16GB: ECUT performs best, solving 22 instances, followed by EEC (20), BW (17), TSI (15) and IC4PN (12). Besides solving most instances, ECUT does so fastest in most cases (one exception is *Kanban*, which only BW and TSI can handle), proving it generally more robust than the other tools.

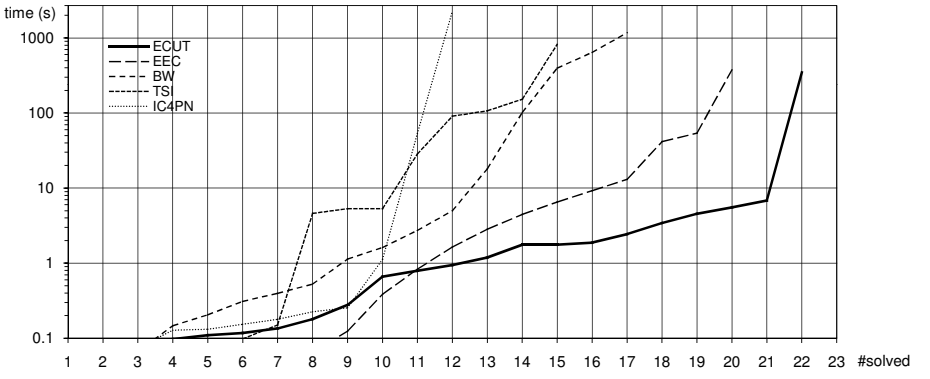


Fig. 4. Comparison of ECUT and the algorithms EEC, BW, TSI and IC4PN for the 23 Petri net benchmarks. Entry (n, t), for a number n and a time t, indicates that it took time t to solve the n easiest instances for the algorithm indicated by the curve.

6.2 Boolean Program Reachability

Our reachability checker SCUT computes the sets R_n using the symbolic model checker BOOM [3]. Since there is no symbolic backward search engine able to handle Boolean programs of the size we consider, we simplify Algorithm 1: thread **A** merely checks whether there are any candidates; if they don't (seem to) vanish for any n, we consider the run a timeout.

We evaluate our implementation of SCUT using 852 Boolean programs. The programs stem from Linux device driver code and were embedded into a concurrent test environment using the DDVERIFY tool [19]. The programs feature on average about 1000 program locations and 9 shared and 18 local variables. We are not aware of any other tool that can perform even finite-state reachability of concurrent Boolean programs of this size. We therefore only present results obtained with our tool.

Table 2 shows analysis results grouped by their cutoff. SCUT succeeds in 798 of the cases (94%); the remaining 54 examples time out (6%). In all safe examples, the cutoff test alone is sufficient: all candidates disappear eventually. We see that for a vast majority of the examples, the cutoffs are indeed very small and easily within the performance limits of BOOM.

7 Related Work

There is a vast amount of literature on tackling reachability analysis for concurrent software, with or without recursion. We focus on work related to the use of cutoffs, and work related to Petri nets. We believe our work to be the first to combine finite-state forward search, cutoff detection and infinite-state backward coverability analysis in a symbiotic manner.

Table 2. Results of sCUT for the Boolean program benchmarks, grouped by cutoffs. #P: # of programs in group; *Sh*, *Lcs*, *Loc*: avg. # shared variables, local variables, program locations; sCUT: avg. sCUT runtime; *c*: cutoff/#threads until error (? = unknown); t/o: # timeouts.

#P	*Sh*	*Lcs*	*Loc*	sCUT	*c*	Safe	Unsafe	t/o
773	17	8	1170	0.1	1	407	366	0
17	21	22	1139	0.8	2	3	14	0
8	13	26	1131	72.3	3	8	0	0
54	18	31	1267	874.0	?	–	–	54

Cutoffs: Much of the work on verifying concurrent programs using cutoffs restricts communication [6,12]. For example, small constant-size cutoffs are known for ring networks communicating only by token passing [12], and for multi-threaded programs communicating only using locks [17]. These results fail to hold, however, with general shared-variable concurrency, as we consider it. On the other hand, [11] permits communication via guards over shared local variables, but gives rise to cutoffs that are linear in the number of states of the program $\mathbb{P}$ being replicated. Such cutoffs are unacceptable for us, as $\mathbb{P}$ may have millions of states.

Bingham presents a technique for coverability that seems closest to our work [4,5]. Standard finite-state BDD techniques are used to compute, for an instance of size n and in a backward fashion, the set of states that have a path to some set U of "bad" states. If the initial state set is intersected, we have encountered an error. Otherwise, n is increased until some convergence criterion is met. Unfortunately, the method is applied to only one (parametric) Petri net. Also, Bingham does not disclose the experimental values of n at which his algorithm terminates, which might give a clue as to the general scalability of the approach — we have found the cutoff of Bingham's Petri net Bh250 to be very small (see Table 1).

Petri nets: Many data structures and algorithms have been proposed for their efficient analysis and coverability checking [15,10]. Most of these algorithms suffer, however, from an intractable number of vector elements after the translation from (Boolean) programs: one per local program state. Recent work by Raskin et al. has attempted to address the dimensionality problem using an abstraction refinement loop [14], where abstract models of the Petri net under investigation are of lower dimension than the original.

Tools: There are several tools available for the analysis of Petri nets [16]. The Mist tool set [13] implements the *Expand, Enlarge and Check* algorithms due to Geeraerts et al. [15]. Furthermore, Petri net/VASS analysis has been applied to Java programs [9] and Boolean programs [2]. These tools compile their input into an explicit-state representation of the underlying program, which may result in a net with a high number of places. Our experiments indicate that, for the case of Boolean program verification, a symbolic representation is essential.

8 Conclusion

We set out to solve the thread-state reachability problem for replicated finite-state programs efficiently. Our proposal is to exploit the (guaranteed) existence of *thread-state cutoffs*, by analyzing the programs for increasing numbers of thread counts. We have presented a sufficient (but not necessary) condition under which the current thread count is a cutoff, so that no larger thread counts need to be considered. We have shown how to make the algorithm complete, using a backward coverability analysis to rule out the reachability of certain *candidate thread states* that were identified to potentially lead to new thread states. The algorithm returns the set of reachable thread state and the minimum cutoff of the given parameterized family.

We have empirically demonstrated, on a large selection of benchmarks, that cutoffs tend to be small enough in practice to allow our incremental technique to beat various methods based solely on coverability algorithms. Our technique is useful both for general Petri net coverability analysis, and specifically for thread-state reachability analysis in non-recursive Boolean programs run by arbitrarily many threads.

Our method can be seen as an opportunity to shift the burden in solving the parameterized verification problem from heavy-weight unbounded tools to lighter-weight *bounded* concurrency model checkers. This is of utmost value, since efficient bounded tools have recently become available, such as Boom, that can solve reachability queries for non-trivial thread counts.

Future work includes the application of our method to extended types of Petri nets, such as transfer nets, which allow richer inter-thread communication, such as broadcasts (an example is S. German's protocol used in [5]).

References

1. Ball, T., Bounimova, E., Cook, B., Levin, V., Lichtenberg, J., McGarvey, C., Ondrusek, B., Rajamani, S.K., Ustuner, A.: Thorough static analysis of device drivers. In: EuroSys, pp. 73–85 (2006)
2. Ball, T., Chaki, S., Rajamani, S.K.: Parameterized verification of multithreaded software libraries. In: Margaria, T., Yi, W. (eds.) TACAS 2001. LNCS, vol. 2031, pp. 158–173. Springer, Heidelberg (2001)
3. Basler, G., Mazzucchi, M., Wahl, T., Kroening, D.: Symbolic counter abstraction for concurrent software. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 64–78. Springer, Heidelberg (2009)
4. Bingham, J.D.: A new approach to upward-closed set backward reachability analysis. Electr. Notes Theor. Comput. Sci. 138(3), 37–48 (2005)
5. Bingham, J.D., Hu, A.J.: Empirically efficient verification for a class of infinite-state systems. In: Halbwachs, N., Zuck, L.D. (eds.) TACAS 2005. LNCS, vol. 3440, pp. 77–92. Springer, Heidelberg (2005)
6. Bouajjani, A., Müller-Olm, M., Touili, T.: Regular symbolic analysis of dynamic networks of pushdown systems. In: Abadi, M., de Alfaro, L. (eds.) CONCUR 2005. LNCS, vol. 3653, pp. 473–487. Springer, Heidelberg (2005)

7. Cardoza, E., Lipton, R.J., Meyer, A.R.: Exponential space complete problems for Petri nets and commutative semigroups: Preliminary report. In: STOC, pp. 50–54 (1976)
8. Cook, B., Kroening, D., Sharygina, N.: Verification of Boolean programs with unbounded thread creation. Theor. Comput. Sci. 388(1-3), 227–242 (2007)
9. Delzanno, G., Raskin, J.-F., Begin, L.V.: 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)
10. Delzanno, G., Raskin, J.-F., Begin, L.V.: Covering sharing trees: a compact data structure for parameterized verification. STTT 5(2-3), 268–297 (2004)
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. Emerson, E.A., Namjoshi, K.S.: Reasoning about rings. In: POPL, pp. 85–94 (1995)
13. Ganty, P., Begin, L.V., Delzanno, G., Raskin, J.-F.: The MIST2 tool, release 1.0, Université Libre de Bruxelles (June 2009), http://www.ulb.ac.be/di/ssd/pganty/software/software.html
14. Ganty, P., Raskin, J.-F., Begin, L.V.: From many places to few: Automatic abstraction refinement for Petri nets. Fundam. Inform. 88(3), 275–305 (2008)
15. Geeraerts, G., Raskin, J.-F., Begin, L.V.: Expand, enlarge and check... made efficient. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 394–407. Springer, Heidelberg (2005)
16. Heitmann, F., Moldt, D.: Petri net tool database, http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html
17. Kahlon, V., Ivancic, 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)
18. Karp, R.M., Miller, R.E.: Parallel program schemata. J. Comput. Syst. Sci. 3(2), 147–195 (1969)
19. Witkowski, T., Blanc, N., Kroening, D., Weissenbacher, G.: Model checking concurrent linux device drivers. In: ASE, pp. 501–504 (2007)

PARAM: A Model Checker for Parametric Markov Models

Ernst Moritz Hahn[1], Holger Hermanns[1,2], Björn Wachter[1], and Lijun Zhang[3]

[1] Saarland University – Computer Science, Saarbrücken, Germany
[2] INRIA Grenoble – Rhône-Alpes, France
[3] DTU Informatics, Technical University of Denmark, Denmark

Abstract. We present PARAM 1.0, a model checker for parametric discrete-time Markov chains (PMCs). PARAM can evaluate temporal properties of PMCs and certain extensions of this class. Due to parametricity, evaluation results are polynomials or rational functions. By instantiating the parameters in the result function, one can cheaply obtain results for multiple individual instantiations, based on only a single more expensive analysis. In addition, it is possible to post-process the result function symbolically using for instance computer algebra packages, to derive optimum parameters or to identify worst cases.

1 Introducing PARAM

Markov processes are applied in computer science, engineering, mathematics, and biology. In the early design phase of a system or for the sake of robust modelling, it can be advantageous to leave certain aspects unspecified and use symbolic parameters rather than fixed values. We consider *parametric* Markov chains (PMCs) [1], where e.g., certain transition probabilities are symbolic parameters. As a result, the analysis of properties, such as the probability of reaching a set of goal states, yields symbolic expressions [2,3] rather than concrete values. These symbolic expressions are represented by multivariate rational functions, i.e. fractions where numerator and denominator are polynomials in the model parameters. To arrive at these results, PARAM uses efficient techniques to manipulate and represent polynomials combined with dedicated state-lumping techniques. The basic analysis provided by PARAM is the computation of step-bounded and unbounded reachability probabilities for PMCs, but it also can handle several extensions of this analysis, as we will explain below.

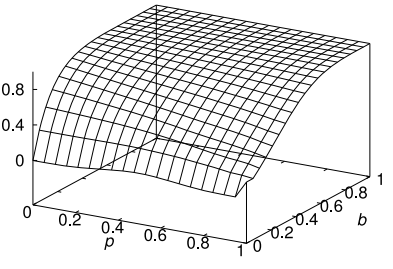


Fig. 1. Crowds Reliability

Consider the Crowds protocol [4,5] where communication is hidden via random routing to protect the anonymity of Internet users. Assume we have n honest Crowd members and m dishonest members. Further, assume that a Crowd member is untrustworthy with probability $b = m/(m+n)$ and denote the probability for random routing (instead of sending directly to the final receiver) by p. We

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 660–664, 2010.

are interested in the probability q that the untrustworthy members observe the original sender more often than any other participant. In practical applications, the Crowds protocol is deployed with many different parameter instantiations, so it is beneficial to explore the resulting range of behaviours with PARAM.

The Crowds protocol can be conveniently modelled in the input language of PARAM, a variation of the language of the PRISM [6] model checker. From the PMC model, PARAM automatically computes a rational function in the parameters of the model. Assume we have $n = 5$ honest members and $r = 7$ rounds of the protocol. Then, the result is a rational function involving multivariate polynomials of degree 21. We can export this function to a computer algebra package for post-processing, e.g. to find optimum parameter settings. In Fig. 1, we plot the dependency of q from the probability b of untrustworthiness of a member and the forwarding probability p. To arrive at this plot, we have evaluated the rational function at 50*60 parameter instances. We could, of course, have solved a total of 3000 non-parametric model checking problems instead to arrive at the same plot (which takes more time). For $n = r = 2$ the formula is:

$$q(p,b) = \frac{p^2b^4 + 2p^2b^3 - 7p^2b^2 + 4p^2b + 12pb^2 - 12pb - 4b^2 + 8b}{4p^2b^2 - 8p^2b + 4p^2 + 8pb - 8p + 4}$$

PARAM can handle the Crowds model up to roughly $n = 10, r = 8$, leading to a state-space of about 2.5 million states.

The method of computing rational functions is inspired by Daws [7], who treats a PMC as a finite automaton and computes the regular expression of its accepted language [8], from which the rational function is obtained. Our method instead works directly on rational functions, and simplifies them on-the-fly. In doing so, we can exploit symmetries, cancellations and simplifications of arithmetic expressions, especially if most of the transition probabilities of the input model are constants. Experimental evidence shows that this results in drastically smaller rational functions during intermediate computations.

2 Architecture

The architecture and components of PARAM are depicted in Fig. 2.

First, the state-space exploration component generates an explicit graph representation from the symbolic description of the model and property. Thereby it leverages property-dependent optimisations to reduce the number of states, in order to accelerate subsequent steps.

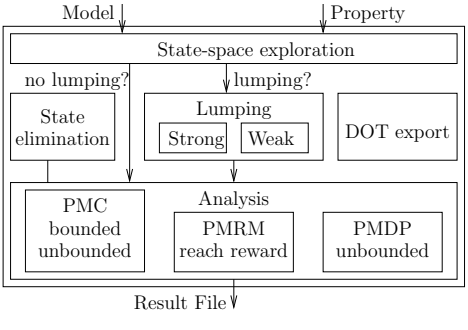


Fig. 2. PARAM Architecture

The lumping [9,10] component leverages bisimulation equivalence to minimise the model. It selects the adequate property-preserving bisimulation relation. The lumping component is

modular and can easily be adapted. It can lead to dramatic speed-ups. For parametric Markov decision processes (PMDPs), lumping maintains only maximal reachability [2,3].

To produce the final result in the form of a rational function, the analysis component uses a variant of state elimination and operations on polynomials. Three dedicated engines cater to specific variants of parametric models: PMCs and extended models with rewards or non-determinism.

3 Selected Features

The algorithmic basis for our tool has been laid out in a previous publication [2] together with a prototypical implementation. PARAM 1.0 [3] improves and enhances this prototype in many aspects. We give an overview of important distinguishing features below.

SPIN-like model exploration. Our prototype suffered from a rather inefficient low-level state-space generation. Using a SPIN-like [11] precompilation technique, we improved the speed of the state-space generation by a factor of about ten. To this end, we first convert the PRISM model into a C++ library, compile it and generate the state-space using the library.

Reward models. In addition to PMCs, PARAM 1.0 can also handle parametric Markov *reward* models (PMRMs) in which states and transitions are additionally equipped with reward structures, and these reward structures can be specified as parameters. For PMRMs, we consider reachability rewards, that is, the accumulated rewards till a certain set of target states is reached.

Nondeterministic models. PARAM 1.0 supports PMDPs, which involve both parametric probabilistic choice and nondeterminism. We are interested in the *maximum* probability of reaching a given set of target states. PARAM first reduces the problem to reachability over PMCs, by encoding the nondeterministic choices as additional parameters, and then submits the problem to the engine for PMCs. Admittedly, this approach does not scale, since the analysis is exponential in the number of nondeterministic states.

Improved data structures. The prototype of PARAM relied on data structures provided by the arithmetic library CoCoALib [12]. PARAM 1.0 instead uses dedicated data structures and a novel, memory-efficient implementation of rational functions, to avoid costly conversions between PARAM and CoCoALib. CoCoALib is still used to cancel rational functions, as algorithms to do so are very complicated to implement in an efficient manner. PARAM 1.0 tries to call the cancellation routine only if cancellation is detected to be possible.

Sharing. PARAM 1.0 uses a global table of rational functions. The same rational function is only stored once. Rational functions attached to state transitions are references into this table. This is advantageous, because experiments show

that rational functions have a tendency of becoming quite large during the state elimination. At the same time, intermediate computations often result in identical rational functions. To exploit sharing, we use a fast hashing mechanism to find such rational functions in the table. This is important, as table lookups are frequent. In the prototype, this feature was not yet implemented.

Value Cache. Arithmetic operations on rational functions are much more costly than for floating-point values. To avoid redundant re-computation, we have implemented a value cache which stores operands and results of operations. Our experiments have shown that, for our current implementation of state elimination, the value cache is only useful during lumping. Because of this, our implementation of rational functions allows to toggle this feature, for different parts of the analysis. This feature existed in the prototype, but is improved in release 1.0.

Lumping Bisimulation. PARAM 1.0 features an efficient implementation of lumping algorithms. Using signature-based lumping [13] is the key to an efficient implementation. Currently, weak and strong bisimulation for PMCs are implemented. The mechanism is written in a modular way, allowing future integration of other lumping techniques into the framework. Since operations on rational functions are rather expensive, this feature is very important in practice. In certain cases, this component can be used for PMDPs.

4 Concluding Remarks

PARAM 1.0 consists of approximately 5000 lines of C++ code, and has been tested on a large number of case studies. It is available for Linux with libc6. The source code is published under the GPL license. The source code and several case studies can be downloaded from `http://depend.cs.uni-sb.de/tools/param`

Acknowledgement. This work is supported by the NWO-DFG bilateral project ROCKS, by the DFG as part of the Transregional Collaborative Research Centre SFB/TR 14 AVACS and the GRK 623 and has received funding from the European Community's Seventh Framework Programme under grant agreement n^o 214755. The work of Lijun Zhang is partially supported by MT-LAB, a VKR Centre of Excellence. Part of this work was done while Lijun Zhang was at Saarland University and Computing Laboratory, Oxford University.

References

1. Lanotte, R., Maggiolo-Schettini, A., Troina, A.: Parametric probabilistic transition systems for system design and analysis. FAC 19, 93–109 (2007)
2. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. In: Păsăreanu, C.S. (ed.) SPIN. LNCS, vol. 5578, pp. 88–106. Springer, Heidelberg (2009)
3. Hahn, E.M., Hermanns, H., Zhang, L.: Probabilistic reachability for parametric Markov models. STTT (2010) doi: 10.1007/s10009-010-0146-x

4. Reiter, M.K., Rubin, A.D.: Crowds: anonymity for web transactions. ACM Trans. Inf. Syst. Secur. 1, 66–92 (1998)
5. Shmatikov, V.: Probabilistic model checking of an anonymity system. Journal of Computer Security 12, 355–377 (2004)
6. 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)
7. Daws, C.: Symbolic and parametric model checking of discrete-time Markov chains. In: Liu, Z., Araki, K. (eds.) ICTAC 2004. LNCS, vol. 3407, pp. 280–294. Springer, Heidelberg (2005)
8. Hopcroft, J.E., Motwani, R., Ullman, J.D.: Introduction to automata theory, languages, and computation, SIGACT News, 2nd edn., vol. 32, pp. 60–65 (2001)
9. Derisavi, S., Hermanns, H., Sanders, W.H.: Optimal state-space lumping in Markov chains. Inf. Process. Lett. 87, 309–315 (2003)
10. Baier, C., Hermanns, H.: Weak bisimulation for fully probabilistic processes. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 119–130. Springer, Heidelberg (1997)
11. Holzmann, G.: SPIN model checker, The: primer and reference manual. Addison-Wesley Professional, Reading (2003)
12. Abbott, J.: The design of CoCoALib. In: Iglesias, A., Takayama, N. (eds.) ICMS 2006. LNCS, vol. 4151, pp. 205–215. Springer, Heidelberg (2006)
13. Derisavi, S.: Signature-based symbolic algorithm for optimal Markov chain lumping. In: QEST, pp. 141–150 (2007)

GIST: A Solver for Probabilistic Games*

Krishnendu Chatterjee[1], Thomas A. Henzinger[1],
Barbara Jobstmann[2], and Arjun Radhakrishna[1]

[1] IST Austria (Institute of Science and Technology Austria)
[2] CNRS/Verimag, France

Abstract. GIST is a tool that (a) solves the qualitative analysis problem of turn-based probabilistic games with ω-regular objectives; and (b) synthesizes reasonable environment assumptions for synthesis of unrealizable specifications. Our tool provides the first and efficient implementations of several reduction-based techniques to solve turn-based probabilistic games, and uses the analysis of turn-based probabilistic games for synthesizing environment assumptions for unrealizable specifications.

1 Introduction

GIST (Game solver from IST) is a tool for (a) qualitative analysis of *turn-based probabilistic games ($2\frac{1}{2}$-player games)* with ω-regular objectives, and (b) computing environment assumptions for synthesis of unrealizable specifications. The class of $2\frac{1}{2}$-player games arise in several important applications related to verification and synthesis of reactive systems. Some key applications are: (a) synthesis of stochastic reactive systems; (b) verification of probabilistic systems; and (c) synthesis of unrealizable specifications. We believe that our tool will be useful for the above applications. GIST is available for download at `http://pub.ist.ac.at/gist`.

$2\frac{1}{2}$-player games. $2\frac{1}{2}$-player games are played on a graph by two players along with probabilistic transitions. We consider ω-regular objectives over infinite paths specified by parity, Rabin and Streett (strong fairness) conditions that can express all ω-regular properties such as safety, reachability, liveness, fairness, and most properties commonly used in verification. Given a game and an objective, our tool determines whether the first player has a strategy to ensure that the objective is satisfied with probability 1, and if so, it constructs such a witness strategy. Our tool provides the first implementation of qualitative analysis (probability 1 winning) of $2\frac{1}{2}$-player games with ω-regular objectives.

Synthesis of environment assumptions. The synthesis problem asks to construct a finite-state reactive system from an ω-regular specification. In practice, initial specifications are often unrealizable, which means that there is no system that implements the specification. A common reason for unrealizability is that

* This research was supported by the European Union project COMBEST and the European Network of Excellence ArtistDesign.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 665–669, 2010.

assumptions on the environment of the system are incomplete. The problem of correcting an unrealizable specification Ψ by computing an environment assumption Φ such that the new specification $\Phi \rightarrow \Psi$ is realizable was studied in [2]. The work [2] constructs an assumption Φ that constrains only the environment and is as weak as possible. Our tool implements the algorithms of [2]. We believe our implementation will be useful in analysis of realizability of specifications and computation of assumptions for unrealizable specifications.

2 Definitions

Game graphs. A *turn-based probabilistic game graph* ($2\frac{1}{2}$*-player game graph*) $G = ((S, E), (S_0, S_1, S_P), \delta)$ consists of a directed graph (S, E), a partition (S_0, S_1,S_P) of the finite set S of states, and a probabilistic transition function δ: $S_P \rightarrow \mathcal{D}(S)$, where $\mathcal{D}(S)$ denotes the set of probability distributions over the state space S. The states in S_0 are the *player*-0 states, where player 0 decides the successor state; the states in S_1 are the *player*-1 states, where player 1 decides the successor state; and the states in S_P are the *probabilistic* states, where the successor state is chosen according to the probabilistic transition function δ. *2-player game graphs* are a special case where $S_P = \emptyset$.

Objectives. We consider the three canonical forms of ω-regular objectives: Streett and its dual Rabin objectives; and parity objectives. The Streett objective consists of d request-response pairs $\{ (Q_1, R_1), (Q_2, R_2), \ldots, (Q_d, R_d) \}$ where Q_i denotes a request and R_i denotes the corresponding response (each Q_i and R_i are subsets of the state space). The objective requires that if a request Q_i happens infinitely often, then the corresponding response must happen infinitely often. The Rabin objective is its dual. The parity objective is a special case of Streett objectives where $Q_1 \subset R_1 \subset Q_2 \subset R_2 \subset Q_3 \subset \cdots \subset Q_d \subset R_d$.

Qualitative analysis. The qualitative analysis for $2\frac{1}{2}$-player games is as follows: the input is a $2\frac{1}{2}$-player game graph, and an objective Φ (Streett, Rabin or parity objective), and the output is the set of states such that player 0 can ensure Φ with probability 1. For detailed description of game graphs, plays, strategies, objectives and notion of winning see [1]. We focus on qualitative analysis because: a) In applications like synthesis qualitative analysis is more relevant: the goal is to synthesize a system that behaves correctly with probability 1; (b) Qualitative analysis for probabilistic games is independent of the precise probabilities, and thus robust with imprecision in the exact probabilities (hence resilient to probabilistic modeling errors). The qualitative analysis can be done with discrete graph theoretic algorithms. Thus, qualitative analysis is more robust and efficient, and our tools implements these efficient algorithms.

3 Tool Implementation

Qualitative analysis of $2\frac{1}{2}$-player games. Our tool presents the first implementation for the qualitative analysis of $2\frac{1}{2}$-player games with Streett, Rabin

and parity objectives. We have implemented the linear-time reduction for qualitative analysis of $2\frac{1}{2}$-player Rabin and Streett games to 2-player Rabin and Streett games of [1], and the linear-time reduction for $2\frac{1}{2}$-player parity games to 2-player parity games of [4]. The 2-player Rabin and Streett games are solved by reducing them to the 2-player parity games using the LAR construction [5]. The 2-player parity games are solved using the tool PGSolver [6].

Environment assumptions for synthesis. Our tool implements a two-step algorithm for computing the environment assumptions as presented in [2]. The algorithm operates on the game graph that is used to answer the realizability question. First, a safety assumption that removes a minimal set of environment edges from the graph is computed. Second, a fairness assumption that puts fairness conditions on some of the remaining environment edges is computed. The problem of finding a minimal set of fair edges is computationally hard [2], and a reduction to $2\frac{1}{2}$-player games was presented in [2] to compute a locally minimal fairness assumption. The details of the implementation are as follows: given an LTL formula ϕ, the conversion to an equivalent deterministic parity automaton is achieved through GOAL [7]. Our tool then converts the parity automaton into a 2-player parity game by splitting the states and transitions based on input and output symbols. Our tool then computes the safety assumption by solving a safety model-checking problem. The computation of the fairness assumption is achieved in the following steps:

- Convert the parity game with fairness assumption into a $2\frac{1}{2}$-player game.
- Solve the $2\frac{1}{2}$-player game (using our tool) to check whether the assumption is sufficient (if so, go to the previous step with a weaker fairness assumption).

The synthesized system is obtained from a witness strategy of the parity game. The flow is illustrated in Figure 1.

We illustrate, how our tool works, on a simple example. Consider the LTL formula $\Phi = GF(g) \wedge G(c \rightarrow \neg g)$, where G and F denote globally and eventually, respectively. The specification says that we want to see infinitely many grants (g), but when we receive a cancel (c) we are not allowed to give a grant. From Φ our tool constructs a deterministic parity automaton that accepts exactly the words that satisfies Φ. The parity automaton is then converted into the parity game shown in Figure 2(a). We use $\Box$ to represent player-0 states and $\Diamond$ to represent player-1 states. environment cannot force the play outside the

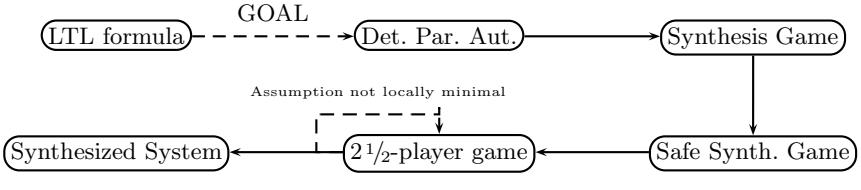


Fig. 1. The flow of the tool for computation of environment assumptions

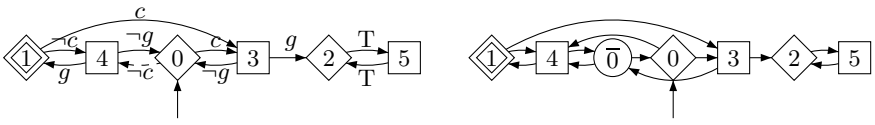


Fig. 2. (a) Parity game with fairness assumption (b) Equivalent $2\frac{1}{2}$-player game

cooperative winning region. Now, we are searching for a locally minimal fairness assumption by reducing a game with fairness assumptions on edges to a $2\frac{1}{2}$-player parity game (see [2]). If the initial state in the $2\frac{1}{2}$-player game is winning with probability 1 for player 0, then the assumption is sufficient. Figure 2(b) illustrates the $2\frac{1}{2}$-player game obtained with a fairness assumption on the edge $(0, 4)$. The ◯ state is a probabilistic state with uniform distribution over its successors. The assumption on edge $(0, 4)$ is the minimal fairness assumption for the example. From this, our tool extract an automaton representing the environment assumption. For the example, we obtain an automaton equivalent to $G(c \rightarrow \neg g) \rightarrow GF(\neg c)$. The tool also constructs a system that implements the specification under this assumption. The constructed system sets g high whenever c is low and vice-versa.

Performance of Gist**.** Our implementation of reduction of $2\frac{1}{2}$-player games to 2-player games is linear time and efficient, and the computationally expensive step is solving 2-player games. For qualitative analysis of $2\frac{1}{2}$-player games, Gist can handle game graphs of size that can be typically handled by tools solving 2-player games. Typical run-times for qualitative analysis of $2\frac{1}{2}$-player parity games of various sizes are summarized in Table 1. The games used were generated using the benchmark tools of PGSolver and converting one-tenth fraction of the states into probabilistic states (further experimental results in [3]). For synthesis of environment assumptions, the expensive step is the reduction of LTL formula to deterministic parity automata. Our tool can handle formulas that are handled by classical tools for translation of LTL formula to deterministic parity automata.

Other features of Gist**.** Our tool is compatible with several other game solving and synthesis tools: Gist is compatible with PGSolver and GOAL. Our tool provides a graphical interface to describe games and automata, and thus can also be used as a front-end to PGSolver to graphically describe games.

Table 1. Runtimes for solving $2\frac{1}{2}$-player parity games

States	Edges	Runtime (sec.)		
		Avg.	Best	Worst
1000	5000	1.17	0.63	1.59
10000	50000	51.43	39.38	62.61
50000	250000	2513.18	2063.40	2711.23

References

1. Chatterjee, K.: Stochastic ω-Regular Games. PhD thesis, UC Berkeley (2007)
2. Chatterjee, K., Henzinger, T.A., Jobstmann, B.: Environment assumptions for synthesis. In: van Breugel, F., Chechik, M. (eds.) CONCUR 2008. LNCS, vol. 5201, pp. 147–161. Springer, Heidelberg (2008)
3. Chatterjee, K., Henzinger, T.A., Jobstmann, B., Radhakrishna, A.: Gist: A solver for probabilistic games. CoRR, abs/1004.2367 (2010)
4. Chatterjee, K., Jurdziński, M., Henzinger, T.A.: Quantitative stochastic parity games. In: Ian Munro, J. (ed.) Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2004, New Orleans, Louisiana, USA, January 11-14. SIAM, Philadelphia (2004)
5. Gurevich, Y., Harrington, L.: Trees, automata, and games. In: STOC '82: Proceedings of the fourteenth annual ACM symposium on Theory of computing, San Francisco, California, United States, pp. 60–65. ACM Press, New York (1982) ISBN: 0-89791-070-2, http://doi.acm.org/10.1145/800070.802177
6. Lange, M., Friedmann, O.: The pgsolver collection of parity game solvers. Technical report, Institut für Informatik, Ludwig-Maximilians-Universität (2009)
7. Tsay, Y., Chen, Y., Tsai, M., Chan, W., Luo, C.: 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)

A NuSMV Extension for Graded-CTL Model Checking

Alessandro Ferrante[1], Maurizio Memoli[2], Margherita Napoli[2], Mimmo Parente[2], and Francesco Sorrentino[3]

[1] Embedded Systems Research Unit, Fondazione Bruno Kessler
ferrante@fbk.eu
[2] Dipartimento di Informatica ed Applicazioni, Università degli Studi di Salerno
{memoli,napoli,parente}@dia.unisa.it
[3] Computer Science Department, University of Illinois at Urbana Champaign
sorrent1@uiuc.edu

Abstract. Graded-CTL is an extension of CTL with graded quantifiers which allow to reason about either *at least* or *all but* any number of possible futures. In this paper we show an extension of the NuSMV model-checker implementing symbolic algorithms for graded-CTL model checking. The implementation is based on the CUDD library, for BDDs and ADDs manipulation, and includes also an efficient algorithm for multiple counterexamples generation.

1 Description and Architecture

In this paper we introduce a new model-checker which is an extension of NuSMV [CCG+02], an efficient and easy to extend re-implementation and integration of the SMV model-checker [McM93, CMCH96]. Our tool implements symbolic algorithms for graded-CTL model checking[1]. Graded-CTL [FNP08, FNP09a] is an extension of CTL with graded quantifiers that allow to reason about either *at least* or *all but* any number of possible futures. For example, the formula $E^{>k}\mathcal{F}(critic1 \wedge critic2)$ expresses that there are more than k possibilities (i.e. k different paths in the Kripke structure modeling the system) to violate the mutual exclusion property. Formulas of these types cannot be expressed in CTL and not even in μ-calculus (though they can be easily reduced, in exponential time, to equivalent *graded* μ-calculus formulas, [KSV02]). Graded-CTL formulas can be used to determine whether there are more than a given number of bad behaviors of a system: this, in the model-checking framework, means that one can verify the existence of a user-defined number of counterexamples for a given specification and can generate them, in a unique run of the model-checker.

Symbolic Model Checking [BCM+90] applied to CTL is known to behave efficiently, especially in hardware verification, and has been widely studied and implemented in a lot of well known model-checkers. In [FNP09b] *symbolic algorithms* to solve the graded-CTL model checking problem are shown. Graded-CTL NuSMV includes a smart implementation of these algorithms based on

[1] For better readability, we call this extension *Graded*-CTL *NuSMV*.

T. Touili, B. Cook, and P. Jackson (Eds.): CAV 2010, LNCS 6174, pp. 670–673, 2010.

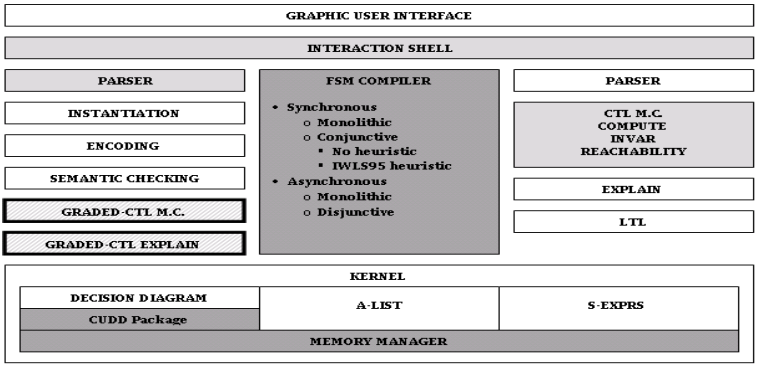


Fig. 1. The architecture of Graded-CTL NuSMV. Light-gray and dark-gray modules are NuSMV modules with some minor and major modifications, respectively. Modules with bold borders are new modules added to support graded-CTL model checking.

the CUDD library [Som05], which is used for an efficient manipulation of sets and multi-sets via *Binary Decision Diagrams* (BDDs) and *Algebraic Decision Diagrams* (ADDs).

NuSMV is a versatile tool, implementing BDD-based and SAT-based model checking, and processing files written in an extension of the SMV language. With this language it is possible to describe finite-state machines (declaration and instantiation of modules and processes are used to describe synchronous and asynchronous composition) and to express specifications in CTL and LTL. NuSMV works either in batch or in interactive mode, with a textual interactive shell.

Our tool preserves the structure and the modularity of NuSMV: each module implements a set of functionalities and communicates with the others via a precisely defined interface. Fig. 1 shows the architecture of Graded-CTL NuSMV pointing out the modified and the completely new modules. The *Interactive Shell*, the *Parser* and the *Compiler* modules are responsible for processing command lines and input files (including syntactic correctness check) and also for building the resulting parse tree and the BDD representation. They have been integrated with the functions and the commands to handle and represent graded-CTL formulas, as well. The *Kernel* module provides the low level functionalities to handle data structures (BDDs, etc.) and memory allocation. It has been modified with a re-implementation and an extension of the cache to store the grading values. The *Model Checking* module is the core of the NuSMV tool: it provides all the functionalities to solve the verification problem. We chose to preserve the structure of NuSMV, and thus we only modified the functions responsible for the invocation of the low level model checking routines, and implemented the graded-CTL model checking algorithms in a new module called *Graded*-CTL *Model Checking*. The implementation of these algorithms required also modifications to the basic operators of the CUDD package. In particular, an implementation of the *AddAndAbstract* operation on ADDs and a bounded leaf-value implementation of the other

operations on ADDs have been included in the package. The latter has led to a considerable improvement in terms of speed and space.

A remarkable feature of the symbolic algorithms we implemented is that they have been explicitly designed to efficiently derive multiple counterexamples for a given path formula, see [FNP09b] for other deeper arguments. In our implementation, we fully exploit this characteristic by using the partial results of the verification phase to derive the needed counterexamples. To do that, the Graded-CTL Model Checking module works interacting with the other new module, *Graded*-CTL *Explain*, responsible for the generation of the counterexamples (see Sect. 2).

Although no absolute criteria are available to evaluate our tool (since, at the best of our knowledge, no tools for similar computations are currently available), the experimental results are very promising. Indeed, our experiments evidenced that no substantial overhead, both in the time and in the number of BDDs, is required to process graded-CTL formulas, with respect to the classical CTL ones, even by increasing the values of the grading constants in the formulas. We are also collaborating with the NuSMV development team to include our extension in the official release. The list of our experiments and the package for graded-CTL can be found at `http://gradedctl.dia.unisa.it`.

2 Counterexamples

A really important feature of Graded-CTL NuSMV is the multiple counterexamples generation. To provide this functionality, the tool has been designed to store the counterexamples in a tree in which each root-to-leaf path represents a distinct counterexample. The computation of this tree is based on three different algorithms used for the evidences generation of $E^{>k}\mathcal{X}$, $E^{>k}\mathcal{G}$, $E^{>k}\mathcal{U}$ respectively. The $E^{>k}\mathcal{X}\psi_1$ case is trivial, so we focus on the others two cases. How to pick the successors and the number of paths to consider are crucial decisions in order to correctly compute the evidences. To compute an evidence-tree for a formula $E^{>k}\theta$ (with either $\theta = \mathcal{G}\psi_1$ or $\theta = \psi_1\mathcal{U}\psi_2$) starting from a state s, our algorithm uses the sets $[E^{>i}\theta] \setminus [E^{>i+1}\theta]$[2] $(0 \leq i \leq k)$ computed during the verification phase. The algorithm starts with $i{=}0$ and computes the set $POST$ of the successors of the state s that are in $[E^{>i}\theta] \setminus [E^{>i+1}\theta]$. Then, recursively generates $i+1$ evidences from each state of $POST$ and store them in a tree rooted in s. At this point, the algorithm halts if $k+1$ evidences have been generated. Otherwise the procedure is repeated for $i+1$. Notice that the evidences are pairwise distinct because the algorithm uses the sets $[E^{>i}\theta] \setminus [E^{>i+1}\theta]$ that are pairwise disjoint. The decision to start with $i = 0$ is motivated by the fact that, in this way, the algorithm generates a *wide* evidence-tree (in opposition to a *tall* evidence-tree that can be obtained by starting with $i = k$) by distinguishing the evidences "as soon as possible". Indeed, in some cases a tall evidence-tree may be constituted by paths which differ only for the number of times that they traverse a self-loop, while a wide tree includes more significative evidences (see [FNP09b] for a deeper discussion).

[2] For a graded-CTL formula φ we denote with $[\varphi]$ the set of states where φ holds. For better readability, we assume $[E^{>k+1}\theta] = \emptyset$.

After that the evidence-tree has been generated, the occurrence of cycles in the paths should be detected and the wrong behaviors should be printed in an intelligible way for the user. In order to detect cycles, a DFS on the tree is performed. The next step is to verify whether the cycle is sink or not: we compute the evidence-tree so that *the node exposing the existence of a cycle has a successor iff the cycle is non-sink*. Finally, the printing phase starts. This phase is important because the counterexamples need to be exposed in such a way that the user can get benefits from the graded logic. In order to have a good balance between information completeness and representation compactness, the tool outputs, for each trace and for each state, a sequential number that identifies the trace and the position within it. Then, for each state, only the *differences* between the previous state are printed. The only exception is for the root of the tree and for those states starting a new branch in the tree. If a group of traces share the same prefix, then it is printed only once and for each new trace only the path from the branch to the leaf is printed. Moreover, the first state of a cycle and the last state of a non-sink cycle are marked with the labels *loop starts here* and *end loop*, respectively. Finally, when a non-sink cycle is found, the trace traversing it only once, is printed, while the traces traversing the cycle more than once are ignored.

References

[BCM+90] Burch, J.R., Clarke, E.M., McMillan, K.L., Dill, D.L., Hwang, L.J.: Symbolic model checking: 10^{20} states and beyond. In: Proc. of LICS 1990, pp. 428–439. IEEE Computer Society Press, Los Alamitos (1990)

[CCG+02] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV Version 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)

[CMCH96] Clarke, E.M., McMillan, K.L., Aguiar Campos, S.V., Hartonas-Garmhausen, V.: Symbolic Model Checking. In: Alur, R., Henzinger, T.A. (eds.) CAV 1996. LNCS, vol. 1102, pp. 419–427. Springer, Heidelberg (1996)

[FNP08] Ferrante, A., Napoli, M., Parente, M.: CTL Model-Checking with Graded Quantifiers. In: Cha, S(S.), Choi, J.-Y., Kim, M., Lee, I., Viswanathan, M. (eds.) ATVA 2008. LNCS, vol. 5311, pp. 20–32. Springer, Heidelberg (2008)

[FNP09a] Ferrante, A., Napoli, M., Parente, M.: Model Checking for Graded-CTL. Fundamenta Informaticae 96(3), 323–330 (2009)

[FNP09b] Ferrante, A., Napoli, M., Parente, M.: Graded-CTL: Satisfiability and Symbolic Model Checking. In: Cavalcanti, A. (ed.) ICFEM 2009. LNCS, vol. 5885, pp. 306–325. Springer, Heidelberg (2009)

[KSV02] Kupferman, O., Sattler, U., Vardi, M.Y.: The Complexity of the Graded μ-Calculus. In: Voronkov, A. (ed.) CADE 2002. LNCS (LNAI), vol. 2392, pp. 423–437. Springer, Heidelberg (2002)

[McM93] McMillan, K.L.: Symbolic Model Checking. Kluwer Academic Publishers, Dordrecht (1993)

[Som05] Somenzi, F.: CUDD: CU Decision Diagram Package, Release 2.4.1 (2005), http://vlsi.colorado.edu/~fabio/CUDD/

Author Index